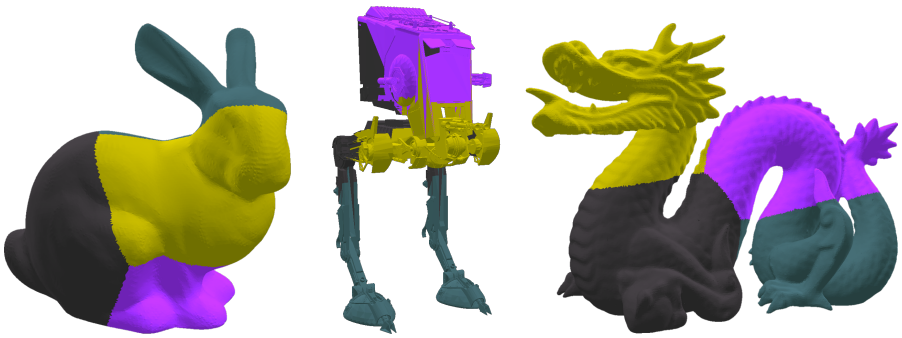
NEW GEOMETRIC ALGORITHMS AND DATA STRUCTURES FOR COLLISION DETECTION OF DYNAMICALLY DEFORMING OBJECTS

David Mainzer



Dissertation, 2015

First printing, October 2015

NEW GEOMETRIC ALGORITHMS AND DATA STRUCTURES FOR COLLISION DETECTION OF DYNAMICALLY DEFORMING OBJECTS

Doctoral Thesis
(Dissertation)

to be awarded the degree of

Doctor rerum naturalium (Dr. rer. nat.)

submitted by

David Mainzer
from Mühlhausen/Thüringen, Germany

approved by the

Faculty of Mathematics/Computer Science and
Mechanical Engineering
Clausthal University of Technology

Date of oral examination
24. September 2015

Chairperson of the Board of Examiners
Prof. Dr. Michael Kolonko

Chief Reviewer
Prof. Dr. Gabriel Zachmann

Reviewer
Prof. Dr. Sven Hartmann

Dedicated to my family.

ABSTRACT

Any virtual environment that supports interactions between virtual objects and/or a user and objects, needs a collision detection system to handle all interactions in a physically correct or plausible way. A collision detection system is needed to determine if objects are in contact or interpenetrates. These interpenetrations are resolved by a collision handling system. Because of the fact, that in nearly all simulations objects can interact with each other, collision detection is a fundamental technology, that is needed in all these simulations, like physically based simulation, robotic path and motion planning, virtual prototyping, and many more. Most virtual environments aim to represent the real-world as realistic as possible and therefore, virtual environments getting more and more complex. Furthermore, all models in a virtual environment should interact like real objects do, if forces are applied to the objects. Nearly all real-world objects will deform or break down in its individual parts if forces are acted upon the objects. Thus deformable objects are becoming more and more common in virtual environments, which want to be as realistic as possible and thus, will present new challenges to the collision detection system. The necessary collision detection computations can be very complex and this has the effect, that the collision detection process is the performance bottleneck in most simulations.

Most rigid body collision detection approaches use a Bounding Volume Hierarchy (BVH) as acceleration data structure. This technique is perfectly suitable if the object does not change its shape. For a soft body an update step is necessary to ensure that the underlying acceleration data structure is still valid after performing a simulation step. This update step can be very time consuming, is often hard to implement and in most cases will produce a degenerated BVH after some simulation steps, if the objects generally deform. Therefore, the here presented collision detection approach works entirely without an acceleration data structure and supports rigid and soft bodies. Furthermore, we can compute inter-object and intra-object collisions of rigid and deformable objects consisting of many tens of thousands of triangles in a few milliseconds. To realize this, a subdivision of the scene into parts using a fuzzy clustering approach is applied. Based on that all further steps for each cluster can be performed in parallel and if desired, distributed to different Graphics Processing Units (GPUs). Tests have been performed to judge the performance of our approach against other state-of-the-art collision detection algorithms. Additionally, we integrated our approach into Bullet, a commonly used physics engine, to evaluate our algorithm.

In order to make a fair comparison of different rigid body collision detection algorithms, we propose a new collision detection Benchmarking Suite. Our Benchmarking Suite can evaluate both the performance as well as the quality of the collision response. Therefore, the Benchmarking Suite is subdivided into a Performance Benchmark and a Quality Benchmark. This approach needs to be extended to support soft body collision detection algorithms in the future.

ZUSAMMENFASSUNG

Jede virtuelle Umgebung, welche eine Interaktion zwischen den virtuellen Objekten in der Szene zulässt und/oder zwischen einem Benutzer und den Objekten, benötigt für eine korrekte Behandlung der Interaktionen eine Kollisionsdetektion. Nur dank der Kollisionsdetektion können Berührungen zwischen Objekten erkannt und mittels der Kollisionsbehandlung aufgelöst werden. Dies ist der Grund für die weite Verbreitung der Kollisionsdetektion in die verschiedensten Fachbereiche, wie der physikalisch basierten Simulation, der Pfadplanung in der Robotik, dem virtuellen Prototyping und vielen weiteren. Auf Grund des Bestrebens, die reale Umgebung in der virtuellen Welt so realistisch wie möglich nachzubilden, steigt die Komplexität der Szenen stetig. Fortwährend steigen die Anforderungen an die Objekte, sich realistisch zu verhalten, sollten Kräfte auf die einzelnen Objekte ausgeübt werden. Die meisten Objekte, die uns in unserer realen Welt umgeben, ändern ihre Form oder zerbrechen in ihre Einzelteile, wenn Kräfte auf sie einwirken. Daher kommen in realitätsnahen, virtuellen Umgebungen immer häufiger deformierbare Objekte zum Einsatz, was neue Herausforderungen an die Kollisionsdetektion stellt. Die hierfür Notwendigen, teils komplexen Berechnungen, führen dazu, dass die Kollisionsdetektion häufig der Performance-Bottleneck in der jeweiligen Simulation darstellt.

Die meisten Kollisionsdetektionen für starre Körper benutzen eine Hüllkörperhierarchie als Beschleunigungsdatenstruktur. Diese Technik ist hervorragend geeignet, solange sich die Form des Objektes nicht verändert. Im Fall von deformierbaren Objekten ist eine Aktualisierung der Datenstruktur nach jedem Schritt der Simulation notwendig, damit diese weiterhin gültig ist. Dieser Aktualisierungsschritt kann, je nach Hierarchie, sehr zeitaufwendig sein, ist in den meisten Fällen schwer zu implementieren und generiert nach vielen Schritten der Simulation häufig eine entartete Hüllkörperhierarchie, sollte sich das Objekt sehr stark verformen. Um dies zu vermeiden, verzichtet unsere Kollisionsdetektion vollständig auf eine Beschleunigungsdatenstruktur und unterstützt sowohl rigide, wie auch deformierbare Körper. Zugleich können wir Selbstkollisionen und Kollisionen zwischen starren und/oder deformierbaren Objekten, bestehend aus vielen Zehntausenden Dreiecken, innerhalb von wenigen Millisekunden berechnen. Um dies zu realisieren, unterteilen wir die gesamte Szene in einzelne Bereiche mittels eines Fuzzy Clustering-Verfahrens. Dies ermöglicht es, dass alle Cluster unabhängig bearbeitet werden und falls gewünscht, die Berechnungen für die einzelnen Cluster auf verschiedene Grafikkarten verteilt werden können. Um die Leistungsfä-

higkeit unseres Ansatzes vergleichen zu können, haben wir diesen gegen aktuelle Verfahren für die Kollisionsdetektion antreten lassen. Weiterhin haben wir unser Verfahren in die Physik-Engine Bullet integriert, um das Verhalten in dynamischen Situationen zu evaluieren.

Um unterschiedliche Kollisionsdetektionsalgorithmen für starre Körper korrekt und objektiv miteinander vergleichen zu können, haben wir eine Benchmarking-Suite entwickelt. Unsere Benchmarking-Suite kann sowohl die Geschwindigkeit, für die Bestimmung, ob zwei Objekte sich durchdringen, wie auch die Qualität der berechneten Kräfte miteinander vergleichen. Hierfür ist die Benchmarking-Suite in den Performance Benchmark und den Quality Benchmark unterteilt worden. In der Zukunft wird diese Benchmarking-Suite dahingehend erweitert, dass auch Kollisionsdetektionsalgorithmen für deformierbare Objekte unterstützt werden.

ACKNOWLEDGEMENTS

Prof. Dr. Gabriel Zachmann

First of all, I wish to express special thanks to my supervisor. He always helped and supported me with good suggestions, comments, and insightful discussions, both while he was with Clausthal University as well as with Bremen University. This has been a considerable contribution to my work.

Prof. Dr. Sven Hartmann

Furthermore, I also would like to express my gratitude to Prof. Dr. Sven Hartmann for accepting the co-advisorship and giving useful comments on this work.

Computer Graphics Group at University of Bremen

Since working alone is impossible nowadays, I would like to thank my colleagues René Weller and Daniel Mohr for the fruitful cooperative work.

Department of Computer Science of the Clausthal University

Also my colleagues from the Department of Computer Science of the Clausthal University supported me on this work, especially Jens Drieseberg, René Fritzsche, Dr. Stefan Guthe and Michael Köster.

My Family

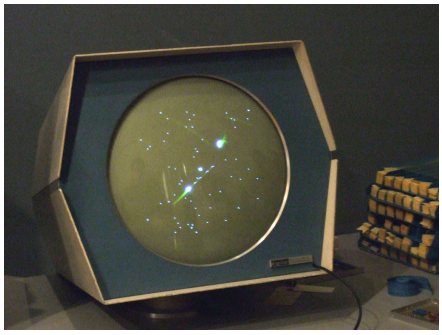
Thanks for giving me my very own computer, and the financial support throughout my study.

Last but not least, my Brother Christoph

I cannot tell you how thankful I am, for your support over all the years. Thank you.

CONTENTS

1 INTRODUCTION



1962 — SpaceWar![1]

2014 — Thief 4[2]

Figure 1.1: Evolution in video games between 1962 and 2014.

Probably there does not exist an area in which a graphics display cannot be used to support the end user. Therefore, computer graphics can be found in many areas such as science, engineering, medical, business, industry, art, education and training. In recent years the degree of optical realism of computer simulations has increased significantly. This important increase is mainly attributable to the computer games scene, where optical realism is basic desire (Figure 1.1 depicts an example of the improvements in computer games over the years).

Nonetheless, it is not the optical component only, that makes virtual environments more realistic, but also naturalistic sound, interaction metaphors and optical correct behavior of the virtual objects within a scene to external influences caused by the simulation or user. We anticipate that a virtual object interacts in the same way like real-world objects do. This basically means that objects, which collide in the virtual world, should act as real items in the real-world. In the case of objects are rigid, then they will repel each other or will break up in its individual components. On the other hand, if objects are soft bodies they will deform. Thus a virtual environment should behave identically. Realizing such realistic behavior may through complex calculations to simulate such a world. Furthermore, virtual environments getting more and more detailed. To accomplish this, a huge amount of primitives or complex functions to modeling objects are used within virtual environments. Recognizing this, new algorithms have to especial tailored to satisfy-more complex virtual environments and extensive requirements going forward.

1 Steve Russell invented SpaceWar! which is one of the first video games.
2 Thief 4, ©Square Enix, 2014

An abstract geometric model usually represents every object in a virtual environment. Most current graphics hardware uses triangles as primitives for rendering. Consequently, a polygonal representation is a natural choice for scenes and objects within this scene and therefore, most real-world models are composed of complexes of triangles [Eri05; SAM09]. A further possibility to represent a virtual object is using a mathematical function, like B-Splines or the most general form Non-Uniform Rational B-Spline (NURBS). Another possibility to create a complex object is the usage of basic primitives (for example spheres, boxes and cylinders) called Constructive Solid Geometry (CSG). It should be noted, that the polygon soup is the most generic polygonal representation of an object [Eri05; VB04].

Such an abstract geometric model can be used to simulate a real-world scene, although the virtual objects will not interact like a real-world object. The reason for this is, that an abstract geometric model has no physical existence, per se. Consequently, virtual objects can pass through each other. It is therefore necessary to prevent virtual objects to interpenetrate.

In fact, we have to detect whether two objects are in contact or not. This technique, to determine if two or more objects are in contact, is called *collision detection*. In order to ensure solidness, and guarantee that objects act, as the user expected, when they come into contact, i. e., no interpenetration, a so-called *collision handling* system is needed. Collision detection is a problem of kinematics, while collision response is a problem of dynamics [Eri05; OD99]. The problem of collision detection has its roots in computational geometry and robotics. Hahn [Hah88] figured out that the collision detection process takes the most computation time in most sequences when the objects are close enough. In most cases the collision detection process takes over 95 % of the computation time. Since this publication faster collision detection approaches have been presented, but exact collision detection remains the bottleneck in most simulations.

The current trend in computer architecture focuses on multi-core Central Processing Units (CPUs) and many-core Graphics Processing Units (GPUs). That is why new algorithms have to fully exploit the potentials of this trend in computer architecture. The beginning of true Multiple-Instruction, Multiple-Data (MIMD) parallelism goes back to Conte di Menabrea [Con43] "Sketch of the Analytical Engine Invented by Charles Babbage" composed in 1843. The first variation of a Single-Instruction, Multiple-Data (SIMD) parallel computer can be traced back to the 1970's [HP12]. Most modern CPU designs, like Intel's Multi Media Extension (MMX) technique or Intel's Streaming SIMD Extensions (SSE) system introduced in 1999, also supports SIMD instructions, in order to improve the performance of multimedia use, primarily video encoding and decoding. Further important parallel computing architectures are Multiple-Instruction, Single-

	SINGLE INSTR.	MULTIPLE INSTR.
SINGLE DATA	SISD	MISD
MULTIPLE DATA	SIMD	MIMD

Table 1.1: Classification of computer architectures using Flynn's taxonomy [Fly72].

Data (MISD)—i. e., the Space Shuttle flight control computers—and Single-Instruction, Single-Data (SISD)—corresponds to the Von Neumann architecture [Neu45]. An overview of common parallel computer classes is shown in Table 1.1.

In 1999 NVIDIA popularized the term GPU, while marketed the GeForce 256. NVIDIA's first technical definition of a GPU was:

> "... a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second."
>
> NVIDIA, August 31, 1999

In previous years, the GPU mainly has been used to transform, light and rasterize triangles in 3 D graphics applications only [Ngu07]. Therefore, at the beginning a GPU was focused on handling graphics primitives such as triangles. In 1997–1998, when Intel introduced the Pentium 2 and NVIDIA the RIVA TNT graphics card, the number of transistors on a GPU overtook the number of transistors on a CPU (transistor counts of about 8 M) [Wil13]. Brodtkorb, Dyken, Hagen, Hjelmervik, and Storaasli [Bro+10] and Owens, Houston, Luebke, Green, Stone, and Phillips [Owe+08] demonstrated that using the GPU instead of the CPU can drastically speedup certain algorithms. The principal reason for this is the massive performance of GPU, compared to the CPU. The GPUs were still growing exponentially in performance due to massive parallelism, while CPUs hit the serial performance ceiling (see Figure 1.2). Therefore, parallelism seems to be a forward-looking technology of increasing performance. Furthermore, data volumes have increased extensively over the last decades and arbitrarily large data sets can be efficiently parallelized described by Gustafson's Law [Gus88]. Understandably, parallelism will only affect the performance of parallel code sections, which leads to the problem, that the serial part of the code becomes the bottleneck [BHS13]. This is often referred to Amdahl's law [Amd67]. These two laws, Amdahl's and Gustafson's Law, are common used to predict or estimate parallel performance. Further information can be found in Section 3.2. Today the GPU is one of the most famous SIMD computing device.

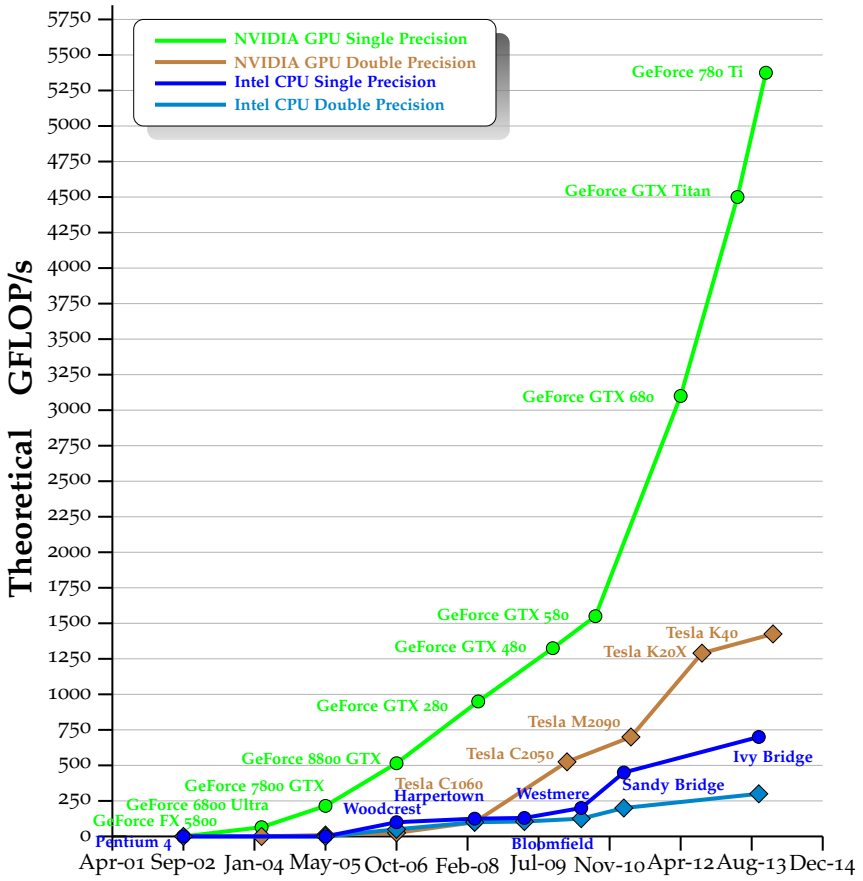


Figure 1.2: Development of the theoretical number of floating-point operations for CPU and GPU over the last years [Nvi14, Section 1.1].

Those key features should be considered when designing a new algorithm, especially when the computation time—needed to solving a problem—is a key specification. However, not all approaches are focused on speed only, in many cases the accuracy is a main issue too. Our novel collision detection approach focuses on accuracy and speed at the same time. To achieve this, we use the GPU as computing device and we are working on the real input data without the need of an acceleration data structure.

1.1 CONTRIBUTIONS

Detecting intersections between moving geometric objects was in most cases a very complicated problem. Therefore, the virtual environment has to determine all colliding objects, or, more precisely, all intersecting primitives in the whole scene to compute a physically plausible collision response. Thus, collision detection is in most complex applications the computational bottleneck [BWK03; BFA02; VT94]. Let us assume that a scene consists of two objects, each of them consisting of n primitives. A straightforward collision

detection approach will test each primitive of one object against all primitives of the other object; this approach is also called brute-force collision detection. This leads to a complexity of $\mathcal{O}(n^2)$ [MP78] (see Chapter 2). A quadratic complexity is completely unsuitable for time sensitive applications and therefore, faster collision detection approaches are needed. However, there are situations, where every primitive collides with all other primitives within the scene (see Figure 2.3) and therefore, $\mathcal{O}(n^2)$ intersections occur. Thus, collision detection approaches—which support every kind of object and object configuration—have a theoretical complexity in the worst-case of $\mathcal{O}(n^2)$.

For example, in games, collision detection and response can be described as "if it looks right, it is right." [Eri05, Section 2.2.2], but other applications, e. g., virtual surgery simulator, path planning and so on, need a much higher accuracy for the collision detection process. Furthermore, to increase the realism of virtual environments soft bodies become more and more important in nearly all applications. Thus, our novel collision detection approach focuses on speed and accuracy as the main issue, while it supports soft and rigid body collision detection.

An overview of the complex area of collision detection is provided in Chapter 2. Since this field has been investigated by research over decades, we are focusing on collision approaches for deformable objects only. Anyway, we also introduce some techniques for rigid body collision detection and some techniques, which are suitable for rigid and soft body collision detection.

As we mentioned before, and Figure 1.2 depicts, parallel computing becomes more and more important. In Chapter 3 a short overview about the field of massively parallel computing is presented. Furthermore, we have a closer look at the GPU, which we use for our approach as computing device.

Most collision detection approaches subdivide the space into regions to faster reject non-colliding parts of an object or whole objects. Chapter 4 gives an overview of commonly used methods for space subdivision. Furthermore, we introduce a clustering approach, which we use for the scene subdivision process. Clustering provides a stable and fast method to create a well-suited partitioning of a scene, even if objects strongly deform and move.

In Chapter 5 we present our novel collision detection approach, which is based on fuzzy scene subdivision. Our method runs entirely on the GPU and therefore, no communication between CPU and GPU is necessary. We use a Principal Component Analysis (PCA) for each cluster of the clustering process to determine the best sweep direction for the Sweep-and-Prune (SaP) algorithm. Furthermore, we tested the performance of our approach in commonly used benchmarks. The

results show that our algorithm is as fast as other state-of-the-art approaches but even more flexible to use more than one GPU only.

This is followed by an technical look at our collision detection approach (see Chapter 6). In this chapter we provide a data flow and sequence diagram. Furthermore, we depict the integration of our approach into a physically based simulation. This serves primarily to check whether behavior and performance of our collision detection approach in a real-world scenario.

In summary, this thesis has the main goal to provide a fast and accurate collision detection for soft bodies which runs in real-time. However, our approach is still not a universal solution that is suitable for every scenario. Thus also other collision detection approaches have the right to exist, e. g., some approaches are especially tailored for collision detection between rigid bodies or focusing on speed only and therefore, they approximate the geometry or use a probability measure to decide if a collision occurs or not.

2

A BRIEF OVERVIEW OF THE COMPLEX AREA OF COLLISION DETECTION

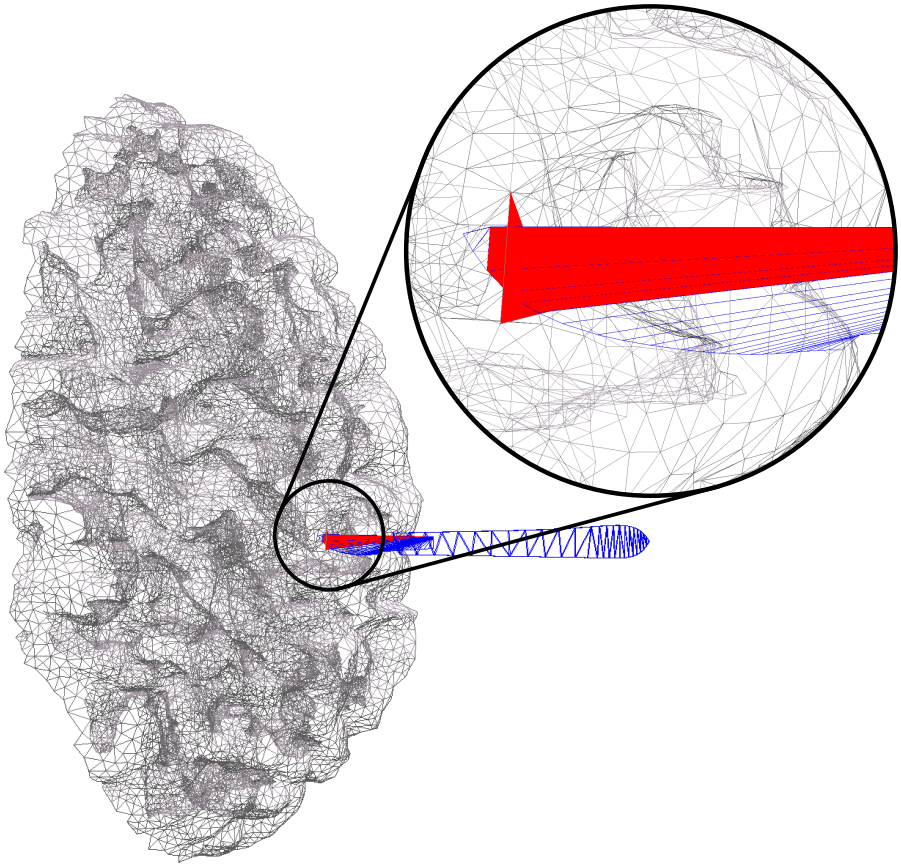


Figure 2.1: Virtual neurological surgery performed with a scalpel (human brain consists of 30 k triangles and scalpel consists of 1 k triangles). Intersecting primitives (triangles) are marked in red.

The area of collision detection has matured from its beginnings around 20-25 years ago to a huge area with a large body of literature, many diverse applications in virtual reality, computational mechanics, medical simulation, games, animations, computational material science, etc., [ES99; Eri05; HKM95; LCN99; Ren+10; YW93], and a large number of methods and principles that were proposed over the course of that time period. Thus it is impossible to discuss all methods in this chapter. Therefore, we will discuss only some often used techniques for spatial partitioning, e. g., spatial hashing, octrees, kd-trees, etc. and for topological methods, e. g., SaP, BVH, and so on. Some of these methods are described in Sections 2.5 to 2.7. Since parallelization getting more and more important we present an overview

of different parallel collision detection approaches in Section 2.9. We close this chapter with some related fields, which can benefit from collision detection data structures also.

2.1 COLLISION DETECTION AND RESPONSE

Collision detection's main focus is to recognize possible interferences between virtual objects, which can move in space or deform in case of deformable geometry. A virtual environment composed of n objects $\{O_0, O_1, \ldots, O_{n-1}\}$, each containing a number of primitives $O_i = \{p_0^i, p_1^i, \ldots, p_k^i\}$, $i = 0, \ldots, n-1$, $k \in \mathbb{N}$. A collision detection approach determines, if objects intersect or not: $O_i \cap O_j \neq \emptyset$, $i = 0, \ldots, n-1$, $j = 0, \ldots, n-1$, and, if self collisions are ignored, $\forall i \neq j$, and, if desired, provide the intersecting primitives, $p_a^i \cap p_b^j \neq \emptyset$.

Let us assume that a scene consists of a given number of objects with a total of n primitives and any one primitive can potentially collide with any other primitive. A straightforward collision detection approach requires $(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$ pairwise primitive intersection tests which is in $\mathcal{O}(n^2)$ [MP78].

In most real-world scenarios few number of primitive intersection pairs occurs and therefore, testing all primitives against each other are unnecessary in many situations. Most collision detection approaches aim to run in real-time or output-sensitive running time [Cla94] and consequently, they have to eliminate as much as possible of the unnecessary $\mathcal{O}(n^2)$ primitive intersection tests to reach this goal. One way of achieving this is to exclude entire objects or large parts of objects to be tested against each other in an early stage of the collision detection process. This process can be thought of as a series of filters, which come once after each other like in a *pipeline* [Zac00] (Figure 2.2 shows an example of a collision detection pipeline). The concept of a linear sequence of modules used for pipelining is commonalty used in computer graphic, e. g., the rendering pipeline, local illumination pipelines, global illumination pipelines and the haptic pipeline [Bar97; Fol+94; MEP92]. The main objective of these pipelining steps is to reduce the number of non-colliding objects or primitives. Like we showed before brute-force collision detection is not a suitable approach because of its high computation time. Therefore, better techniques have been developed to speedup this process. In the following sections we have a closer look at some commonly used techniques in the field of collision detection.

To improve the performance of collision detection the number of objects or primitives tested for intersection has to be reduced and therefore, the collision detection process is often subdivided into two phases: the *broad-phase* (or so-called *n-body processing*) and the *narrow-phase* (or so-called *pair processing*) [Eri05].

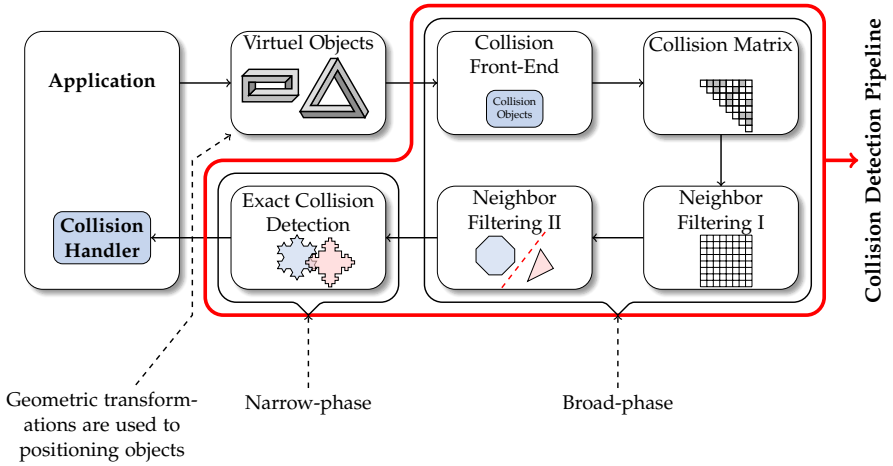


Figure 2.2: An example of a collision detection pipeline with some filtering modules [Wel12; Zac01].

The main focus in the broad-phase of the collision detection is to identify objects or groups of primitives that may be colliding and exclude as rapidly as possible those groups of primitives or whole objects that definitely are not intersecting. The result of this phase is a list of all pairs of potentially colliding primitive groups or objects, the so-called Potentially Collision Set (PCS). The PCS has to be checked for intersections in a second phase, the narrow-phase. Within the narrow-phase some filters can be used to further speedup the collision process [Zac01].

There are many physics simulation libraries, like Bullet [Cou12], Unity3D [Cre10], Unreal Engine [Int12; Moo12], Open Dynamics Engine [Dru+10] and Simulation Open Framework Architecture (SOFA) [Fau+12], which implements different modules and techniques in the collision detection pipeline to eliminate non-colliding parts or objects as early as possible. It should be noted that the collision detection process is only a small part within those physics engines and therefore, implementing a fast and accurate collision detection is not the main focus in most cases.

There are a few requirements, which affect the design of a collision detection pipeline and thereby, fundamentally the collision detection algorithm. The mainly used factor to differentiate collision detection algorithms is the support of rigid bodies and/or deformable objects. Basically all collision detection algorithms, which support deformable objects, can handle collisions between rigid bodies also. However, collision detection approaches focus on rigid bodies only will in any case outperform approaches, which support both types of models.

Another important design factor, while creating a new collision detection pipeline, are the underlying geometric acceleration data structures, which usually are set up in a pre-processing step. Collision

detection algorithms for deformable objects normally require other data structures or they have to implement additional modules in the collision detection pipeline to ensure that used data structures are valid after a simulation step (see Section 2.3).

Furthermore, the underlying abstract geometric model, to represent the objects in a virtual environment, can also be used to classify collision detection approaches. Most approaches use a polygonal representation, e. g., triangles or quads and therefore, many polygonal-based collision detection approaches exist. An efficient algorithm for creating tight-fitting Oriented Bounding Boxes (OBBs) was presented by Gottschalk, Lin, and Manocha [GLM96]. In the initial phase they build up a hierarchical representation of models using OBB-trees. Within collision detection process they use a *separating axis* theorem to check whether or not two OBBs intersect. But the only optimal solution for OBB computation is in $\mathcal{O}(n^3)$ and very hard to implement [ORo85]. Curtis, Tamstorf, and Manocha [CTM08] introduced *Representative-Triangles* to improve the performance of collision handling. A Representative-Triangle is a standard geometric triangle with additional mesh feature information. Furthermore, they can combine their approach with existing BVHs and their approach supports both inter-object and self-collision detection. For a fast continuous collision detection approach Redon, Kheddar, and Coquillart [RKC02] combine Interval Arithmetic (IA) and hierarchies of OBBs. Tang, Curtis, Yoon, and Manocha [Tan+09] combine different techniques to improve the culling efficiency. To cull large regions they use a novel formulation for continuous normal cones. To remove all redundant elementary tests between non-adjacent triangles they introduced so-called *Procedural Representative Triangles*. Other approaches support tetrahedral meshes, which are commonly use to represent volumetric deformable models and therefore, collision detection approaches for this data structure exists. Teschner, Heidelberger, Müller, and Gross [Tes+04] and Teschner, Heidelberger, Müller, Pomerantes, and Gross [Tes+03] developed an algorithm, which use a hash function for compressing a regular spatial grid. They make use of optimized parameters, such as hash function, hash table size and spatial cell size to speedup their collision detection approach. Special collision detection approaches have been developed even for the more rare object representations, like using a mathematical function, B-Splines or the most general form NURBS [GGK06; LG98; PG03].

Lau, Chan, Luk, and Li [Lau+02] described a method, which combines an Axis Aligned Bounding Box (AABB) hierarchy with a bitfield. This approach supports both inter-collision and self-collisions of deformable objects. Page and Guibault [PG03] introduced an approach, which creates OBBs on the fly using the surface control points. Weller and Zachmann [WZ09] introduced a novel data structure for rigid body collision detection, which they called Inner Sphere Tree

(IST). The main idea behind ISTs is to bound objects from the inside with a set of non-overlapping spheres. A key feature of this approach is the possibility to compute the penetration volume, which is related to the water displacement of the overlapping region. Weller, Frese, and Zachmann [WFZ13] presented a parallel collision detection approach for rigid bodies, which has a linear worst-case running time for arbitrary 3 D objects. Therefore, they use the ISTs to fill the objects with spheres. Furthermore, they use a hierarchy of grids to decide if two spheres are close together or not. Because of the fact that spheres within an object do not overlap and they have minimum radii, only a known number of spheres can overlap a cell of the grid. This enables the possibility to predefine the maximum number of spheres within a cell and thus the total maximum duration of the collision detection process. A novel GPU-based collision detection method for deformable parameterized surfaces was introduced by Greß, Guthe, and Klein [GGK06]. They build up an AABB hierarchy from the geometry image for every object individually. The whole computation process is outsourced to the GPU. Weller, Klein, and Zachmann [WKZ06] proposed a model to estimate the expected running time of a collision detection process using an AABB hierarchy.

Creating a complex object from basic primitives (for example spheres, boxes and cylinders) is called CSG. Therefore, Tilove [Til84] use primitive redundancy and spatial localization to reduce the CSG-tree efficiently. Su, Lin, and Yen [SLY96] convert the CSG object model into a Bounding Volume (BV) representation and later use this representation in the collision detection process. Both models are integrated in a hybrid model. A Divide-and-Conquer (DAC) approach is used to identify possible colliding regions in a fast way in combination with an adaptive BV selection strategy. Su, Lin, and Ye [SLY99] described a collision detection approach for CSG, which mainly takes advantages of the CSG Divide-and-Conquer paradigm. In addition to that, they use efficient distance-aided collision detection for convex BVs. Kim, Oh, Yoon, Kim, and Elber [Kim+11] use a hierarchy of Coons [Coo67] patches to approximate the BVH of free-form NURBS surfaces.

Another important model representation is a set of points, also called point clouds, which has become more and more popular in the last years. Cheap 3 D scanning devices, like Microsoft's Kinect [Iza+11] or a simple webcam [PRD09], are the reason for this. Furthermore, nearly everyone owns a webcam and thus a 3 D scanner is widely available [RHL02]. Currently Google is developing a tablet with a 3 D scanning software, called Project Tango. Furthermore, point clouds are well-suited for represent complex models [BWG03; KZ04b; KZ04c; Pfi+00; RL00; Zwi+02]. Klein and Zachmann [KZ04a] presented the first approach for collision detection of point clouds. Their time-critical collision detection approach detects intersection

of the underlying implicit surfaces. Therefore, they create a hierarchy containing of the sample points beside with spheres covering a part of the scanned surface. Klein and Zachmann [KZ05] proposed an approach to reconstruct the intersection curve between two point clouds. Thus, they use interpolation search along the shortest path within a proximity graph. Figueiredo, Oliveira, Araújo, and Pereira [Fig+10] introduced an efficient collision detection approach for point clouds. Therefore, they divide the scene graph into voxels. For every voxel they use an R-tree hierarchy of AABBs for fast collision queries. Pan, Chitta, and Manocha [PCM11] compute the collision probability for every data point. Furthermore, they use a stochastic traversal of BVHs to accelerate the computation. Pan, Sucan, Chitta, and Manocha [Pan+13] introduced an efficient approach, which can handle large sensor generated data sets of point clouds received at real-time rates. Therefore, they use a high-dynamic AABB tree as broad-phase culling technique and use an octree, holding the point cloud data, as proximity data structure. Radwan, Ohrhallinger, and Wimmer [ROW14] proposed a collision detection approach for point clouds, which use a screen-space representation for point clouds. Because the underlying surface is 2 D, they reduced the 3 D point clouds into a number of thickened layered depth images.

All these features, like support of rigid bodies or soft objects, the underlying geometric acceleration data structures, or the underlying abstract geometric model to represent the objects, can be used to classify collision detection algorithms. However, there exists further classification features for collision detection, i. e., accurate or approximate, off-line or real-time, CPU-, GPU-based or hybrid, hierarchical or non-hierarchical methods.

Nevertheless, it must be mentioned that the theoretical complexity of most collision detection approaches is in the worst-case in $\mathcal{O}(n^2)$, if no constraints for the used objects or object configuration exists. Such a worst-case scenario is the collision between two Chazelle[1] polyhedra shown in Figure 2.3.

2.1.1 *Collision Detection Queries*

Interference detection or *intersection testing* problem is the most evident collision detection query. Therefore, the collision detection algorithm answering the Boolean question: for a given configuration; do object A and B interpenetrate or not. The advantages of this query lie in the fact that it is fast and easy to implement. Thus, it is used in most collision detection approaches. However, for deformable collision detection a Boolean result is not enough and all parts, which are in contact or interpenetrate must be detected. In Section 2.3 we discuss

1 Chazelle [Cha84] showed that a Chazelle polyhedron can not be partitioned into fewer than $\Omega(n^2)$ convex parts.

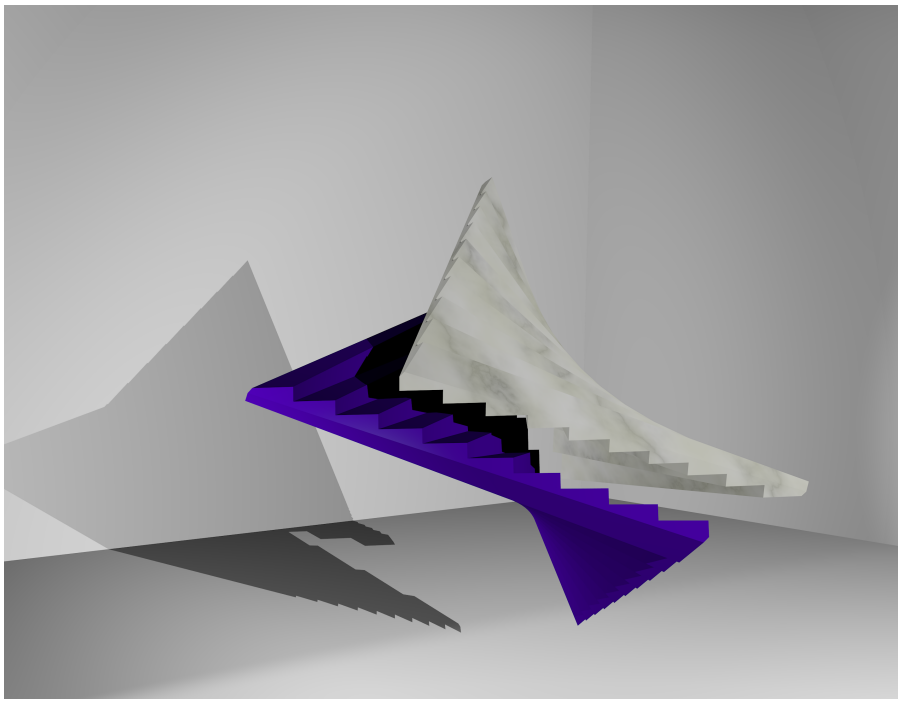


Figure 2.3: One example of a worst-case for collision detection algorithms is the intersection of two Chazelle polyhedra.

why we need to determine *all* contact points to deal with intra- and inter-object collisions in a correct way.

In principle, *approximate queries*—the quality of the answer is only required to be accurate up to a predefined margin of error—are nearly in all cases easier to handle than *exact* collision detection queries. Most computer games are using approximate queries but in the most critical applications, e. g., virtual surgery and physically correct simulations exact collision detection queries are required.

In the case of an intersection, some applications require more information, e. g., *penetration depth*, colliding primitives and/or *intersection volume*. If objects are not intersecting than some applications need *closest points* between two objects A and B. It should be noted that the closest point is not necessarily unique; there can be an infinite number of closest points. In dynamic scenes sometimes the next time of collision is needed, called the Time of Impact (ToI). Many simulations use the ToI to control the time step size in their simulation cycle [Eri05].

2.1.2 *Further Information on Collision Detection*

Since collision detection is a fundamental technique in every physical based simulation or any simulation, where interactions between objects are important, many researchers have investigated it over decades. Naming and discussing of all existing approaches lie far

beyond the scope of an overview. However, we wish to provide a list of the most important publications in this area, but without raising a claim to completeness: the most cited books in the widely field of collision detection are Ericson [Eri05], Van Den Bergen and Bergen [VB04], Eberly [Ebe03], Coutinho [Cou01], Zachmann and Langetepe [ZL03], Overmars [Ove88], Baraff and Witkin [BW92], Foley, Van Dam, Feiner, Hughes, and Phillips [Fol+94] and furthermore, their are many surveys available also, Lin and Gottschalk [LG98], Jiménez, Thomas, and Torras [JTT01], Kockara, Halic, Iqbal, Bayrak, and Rowe [Koc+07], Avril, Gouranton, Arnaldi, et al. [A+09], Teschner, Kimmerle, Heidelberger, Zachmann, Raghupathi, Fuhrmann, Cani, Faure, Magnenat-Thalmann, Strasser, and Volino [Tes+05].

2.2 DEFORMABLE OBJECTS

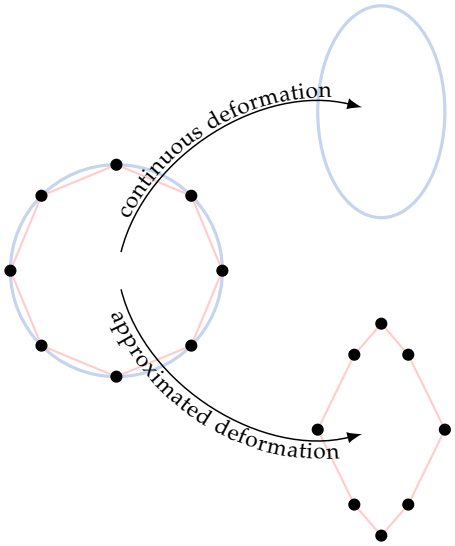


Figure 2.4: The blue circle represents the continuous connected surface of a model $M \in \mathbb{R}^2$. The black dots are points on the surface of the model M, the *material coordinates*, which are connected by a discrete polygonal representation (red lines).

A deformable model is defined by an initial state, often called as rest or equilibrium configuration. This initial state represents the surface of an object *without* the influence of any external forces, like cavitation, inflation pressure or wind. Furthermore, a deformable model consists of a set of material parameters. These parameters define how the model deforms, under the influence of external forces. We define the initial state of a model as a continuous connected surface $M \in \mathbb{R}^n$. All coordinates $m_i \in M$, with i is the index of coordinate, are points of the object, represented by M, and they are called *material coordinates*. In the discrete case—e. g., polygonal models or CSG—M

consists of a fixed number of points that samples the initial state of the model.

When an external force is applied on a deformable model some material coordinates m_i may change their location and move to a new position $def(m_i)$. In the case of a discrete representation of the model only the points representing the surface are moving under the forces (see Figure 2.4). Because the function $def(m_i)$ defines a new location for all material coordinates m_i, $def(m_i)$ can be seen as a vector field defined on M. Another possibility is to define the movement of the material coordinates m_i by a displacement vector field $pos_{DV}(m_i) = def(m_i) - m_i$. For further information on definition of deformable objects and the influence of external forces we want to refer to [Nea+06; Ter+87].

Cloth simulation [Ter+87; Wei86] is a typical application of deformable collision detection. Therefore, many approaches, which are especially well-tailored for cloth simulation have been developed. Lafleur, Magnenat-Thalmann, and Thalmann [LMT91] use a thin force field around each object within the scene to detect collisions. Provot [Pro97] divides the piece of cloth recursively in zones imbricating with each other. In every iteration of the simulation, the BV for each zone is computed. Furthermore, they use a BV-tree to eliminate quickly all collision tests. For cloth simulation Bridson, Fedkiw, and Anderson [BFA02] use a voxel based collision detection approach introduced by McNeely, Puterbaugh, and Troy [MPT99].

2.3 DEFORMABLE VERSUS RIGID BODY COLLISION DETECTION — COLLISION, SELF-COLLISION AND RESTING CONTACT —

The focus of this dissertation lays on accurate collision detection between many deformable objects in an extremely dynamic environment. Nevertheless, we want to show up some key differences between the two main categories—deformable vs. rigid body—of collision handling. Since collision handling for rigid bodies has been researched for almost three decades, it has been well-investigated, but collision handling for deformable objects introduces new problems.

Soft collision detection needs to determine all contact points to create a realistic and correct simulation including those due to self-collisions, whereas collision detections for rigid objects are ignoring self-collisions in most cases. This is one of the main differences between rigid and soft collision detection as well as the underlying acceleration data structure.

Rigid body collision detection mostly uses sophisticated and most often time consuming data structures, which are created once a time at the beginning of the collision handling process. This data structures allows efficient and fast collision detection queries within the simulation at different timestamps. Using these data structures for

deformable collision detection introduce the problem that some parts of a soft body can deform and therefore, most acceleration data structures have to be updated in every simulation step. Depending on the kind of deformation an update of the underlying acceleration data structure can be quite extensive.

Additionally, some rigid body simulations only need one pair of contact points, which improve the performance of the collision handling again. However, a simulation with support for deformable bodies needs to determine *all* contact points to deal with intra-object and inter-object collisions appropriately. Only when all contact points are taking into account, the situation of resting contact between deformable parts of the object and/or different objects can be solved in a physically correct or plausible way [BFA02].

2.4 ANIMATION

> "There is no particular mystery in animation ... it's really very simple, and like anything that is simple, it is about the hardest thing in the world to do."
>
> Bill Tytla, Walt Disney Studios, 1937

An animation introduces a new variable to the object representation, the time parameter. This parameter can be interpreted as a function, which changes the position of an object or a set of points over the time. For the collision detection process we are only in parameters, which can change the geometry representation. There are two common ways to influence the object representation:

- Scene graphs are generally used for rigid body simulations and therefore, altering the transformation in a transform node of the scene graph is a common way to animate the object. This type of motion is often referring to as *placement change*. Placement change uses three types of transformations: translation, rotation and non-uniform scaling on (part of) the model.
- Changing the underlying geometric object directly is another way to animate a model. This type of motion is often referring to as *deformation*. This type is most commonly used for polygonal objects, which can change its shape, like fluids, cloth, or skin.

A scene graph is a hierarchical structure of groups of objects. Every group of objects has their own local coordinate system, which is placed in their parents' coordinate system. With this technique a complex object can be generated easily. In most cases a scene graph is a Directed Acyclic Graph (DAG). A DAG is a directed graph with no *directed* cycles, but a single node can have multiple parents. A scene graph has typically two types of nodes: grouping node and leaf node.

- A *grouping node* combines all child nodes as a subgraph. Therefore, a group node represents the union of all objects hold by its children. Group nodes include all kind of transformations and state switching, e. g., material properties, Level-of-Detail, user defined properties, and more.

- A *leaf node* is rendered and all operations in the tree before are effected to this node. The node can include objects, sounds, lights, camera and many more.

Till today, most placement changes, we can find in interactive 3 D simulations, are rigid motions but we are focus on deformable objects only and therefore, rigid motion is beyond the scope of this work. For further information we refer to Ericson [Eri05] and Van Den Bergen and Bergen [VB04].

Computer graphics researcher investigated the physically motivated deformation of models over three decades. Lasseter [Las87] discussed the squash and stretch principle, which can be occur if an object is moving. Squash and stretch is important in many parts of animation, e. g., facial animation, muscle simulation, and so on. In the same year Terzopoulos, Platt, Barr, and Fleischer [Ter+87] introduced differential equations that model the behavior of deformable objects as a function of time. Solving these differential equations create a physically plausible and realistic animation. In the years that followed, more and more researchers have focused on the topic of visually and physically plausible animation of deformable objects and fluids. A high fidelity approximation of material properties such as plasticity, elasticity, or flexibility approach, which is physically motivated, has been presented by [JP99; ST92; TF88]. A physically plausible behavior of an object within a scene makes interactions with the object much easier. The interactor can predict the effect of an interaction and therefore, planning a sequence of impulse to interact with the environment is possible.

For non-physically motivated geometry representations, like parametric curves and surfaces, and free-form deformations, we refer the interested reader to following approaches: Sederberg, Cardon, Finnigan, North, Zheng, and Lyche [Sed+04] and Sederberg, Zheng, Bakenov, and Nasri [Sed+03] use t-splines, which are a generalization of non-uniform B-spline surfaces. Llamas, Kim, Gargus, Rossignac, and Shaw [Lla+03], Milliron, Jensen, Barzel, and Finkelstein [Mil+02], and Spillmann, Tuchschmid, and Harders [STH13] presented some geometric warps and deformation approaches, which are independent of geometric model representations. Botsch and Kobbelt [BK05] developed a parallel space deformation technique using GPU. Furthermore, methods based on differential surface properties have been developed by [BK04; IMH05; Nea+07; Sor+04; Yu+04]. For deforma-

tion in character animation, especially character skinning reference is made to [JT05; KJP02; McA+11; WP02].

For a physically motivated geometry representation different approaches exist. Most researchers use a simple *explicit* Finite Element Method (FEM), because this method is easy to understand and is straightforward to implement [Deb+01; Mül+02; OH99]. A physically-based character skinning approach was presented by Deul and Bender [DB13].

For the sake of completeness, we want to give a coarse overview of some other methods used to simulate deformable motion [Nea+06, Section 1]:

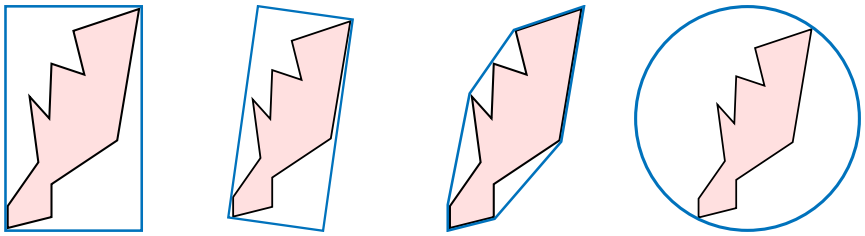
- Lagrangian Mesh Based Methods
  - Continuum Mechanics Based Methods
  - Mass-Spring Systems
- Lagrangian Mesh Free Methods
  - Loosely Coupled Particle Systems
  - Smoothed Particle Hydrodynamics (SPH)
  - Mesh Free Methods for the solution of Partial Differential Equations (PDEs)
- Reduced Deformation Models and Modal Analysis
- Eulerian and Semi-Lagrangian Methods
  - Fluids and Gases
  - Melting Objects

The deformation process of the objects generally is duty of the underlying physics engine and therefore, not focus of this work. Most physics engines are using a collision detection system to decide if the current state of the virtual world is valid or not. Therefore, we have integrated our collision detection approach in Bullet, a commonly used physics engine (see Section 6.5).

2.5 BROAD-PHASE COLLISION DETECTION

The focus in the first part of the collision detection pipeline—the *broad-phase*—is to eliminate as much as possible of the false-positive inter-object collisions from further investigation (see Figure 2.2). To achieve this, typically the geometry of an object is fully encapsulated by a Bounding Volume (BV). A BV has a much more simple shape, which results in a more cheaper overlap test, than testing the whole complex geometry they girdle (few examples of BVs are shown in Figure 2.5). We discuss the types of BVs more in detail in the Section 2.6.

Most objects consist of thousand or even millions of primitives, testing every primitive against each other is in almost every case to expensive. The reason to use a BV is that testing two BVs for overlap can be done very quickly.



(a) AABB (b) OBB (c) Convex Hull (d) Sphere

Figure 2.5: Typical types of BVs

A brute-force implementation of broad-phase—in order to determine whether some BVs overlap or not—will test a BV of one object against all the others' BVs in the scene. This leads to $(n-1)+(n-2)+\ldots+1 = \frac{n(n-1)}{2}$ pairwise BV intersection tests, which has a complexity of $\mathcal{O}(n^2)$, where n is the total number of BVs in the scene. The Axis Aligned Bounding Box (AABB) is one of the most commonly used BV [Eri05]. Woulfe, Dingliana, and Manzke [WDM07] introduced a completely hardware-based broad-phase collision detection approach on Field-Programmable Gate Array (FPGA) using AABBs, but even this full hardware-based approach has a complexity of $\mathcal{O}(n^2)$ and therefore, it is limited to a small amount of AABBs. As early as 1981, Edelsbrunner and Maurer [EM81] have proven that the optimal algorithm to find all intersections of n AABBs has a complexity of $\mathcal{O}(n \cdot \log^2 n + k)$ where k is the number of intersecting objects, or AABBs respectively.

More efficient algorithms, such as spatial subdivision or topological methods, achieve an average complexity of $\mathcal{O}(n \cdot \log n)$, but worst-case complexity is still in $\mathcal{O}(n^2)$.

Spatial partitioning divides space into regions and verified if the BV of two or more objects overlap the same region. Afterwards, these objects, which are in the same region, are passed to the second phase of the collision detection pipeline, the narrow-phase. Grids, trees and spatial sorting are the commonly used spatial partitioning methods. Some examples for these spatial partitioning methods are regular grids, spatial hashing technique [Mir97; PKS10], octrees [PML97; Zho+11], kd-trees [Ben75; BF79; Zho+08], BucketTree [GDO00], R-trees [Gut84; HKM95] and Binary Space Partitioning (BSP) [FKN80]. Under certain conditions spatial partitioning, because of their static nature, can arise inefficiency especially regarding the use of dynamic scenes. This can be because objects are vary greatly in size or an improperly tuned region size is used. For simulations, where the objects are relatively close in size, and their possible bounds are well-

defined, spatial grids can be an excellent and fast choice. Uniform grids, for example, can be updated very fast and this spatial partitioning method is especially well-suited to be parallelized. Le Grand [Le 07] presented a parallel approach using a uniform grid for broad-phase collision detection for particles, while exploiting efficient update and access of uniform grids. Pabst, Koch, and Straßer [PKS10] extended the approach described by Le Grand [Le 07] and presented a hybrid CPU/GPU collision detection for deformable surfaces with self-collision. Another commonly used spatial partitioning method is BSP-tree. Thibault and Naylor [TN87] use BSP-trees to represent an exact representation of arbitrary polyhedra of any dimension. Therefore, they use Boolean expressions on boundary representation as a CSG tree and producing a BSP-tree as the result. Chrysanthou and Slater [CS92], Snyder and Lengyel [SL98], and Torres [Tor90] elaborated different strategies to update spatial data structures. However, often a periodic reconstruction of the tree is required because in highly dynamic scenes the tree may degenerate after several simulation steps. Luque, Comba, and Freitas [LCF05] use an advanced BSP version, called Semi-Adjusting BSP-tree, to subdivide the scene. Their approach use several strategies to alter cutting planes, and defer updates based on the restructuring cost. With this technique they can handle even highly dynamic scenes without a complete restructuring of the BSP-tree.

In comparison with spatial partitioning, *topological methods* are based on the relative position of the objects to each other. This means that objects, which are far away, are not checked for intersection. Baraff [Bar92] introduced one of the most common used topological method, called *sort and sweep*. Cohen, Lin, Manocha, and Ponamgi [Coh+95] referred to this technique as Sweep-and-Prune (SaP) and this designation became generally accepted by most researchers. This approach projects the boundary of the objects' BV on one or more axes. This is followed by an overlap test for all BV intervals on all projection axes. An overlapping of objects BV intervals on all axes means, that their BV intersect and these pairs have to be passed to the narrow-phase. The worst-case complexity of this approach is in $\mathcal{O}(n \cdot \log n + k)$, where n is the number of BVs and k is the amount of overlapping BVs [Bar92; Lin93]. Like for all other approaches more and more parallel solutions have been developed. Avril, Gouranton, and Arnaldi [AGA10] introduced a multi-threading version of the SaP algorithm. They use the three coordinate axes as projection plan. Liu, Harada, Lee, and Kim [Liu+10] perform a parallel SaP using Compute Unified Device Architecture (CUDA). Furthermore, they use PCA to determine the best sweep direction for the SaP algorithm, additionally they use spatial subdivision to reduce false-positive overlaps once more. Shellshear [She14] use a multi-threading 1 D version of the SaP algorithm for collision detection for deformable

cables. Their approach is especially well-suited for cables with a huge amount of moving segments. The Section 2.7 describes the SaP approach in greater detail.

The quality and the time needed for the broad-phase collision detection process depends on the used algorithm. In this context, quality means how many pairs of the PCS, detected by the algorithm, are passed to the exact collision detection phase, although there is not an intersection between them. Zachmann [Zac01] pointed out that the brute-force approach—testing $\frac{n^2+n}{2}$ BVs—performs very well, because of its very small constant factor. The brute-force approach reached its quadratic running time only for many more than 100 objects in a scene. This may explain why most researchers focus on improving and accelerating the narrow-phase. The following section gives a detailed description of the narrow-phase.

2.6 NARROW-PHASE COLLISION DETECTION

Most introduced acceleration data structures, like BVs, BSPs, kd-trees (see Section 4.2 for more details), et cetera can be used to accelerate both broad- and narrow-phase processing. In the broad-phase processing, acceleration data structures are used to represent the whole object in a scene and to cull away entire objects or groups of objects, which are far away. On the other hand, in the narrow-phase the acceleration data structures represent parts of a complex object, to perform fast and cheap overlap tests, before performing expensive primitive-primitive intersection tests. The focus in the broad-phase is on creating a list of possible colliding object pairs, whereas the focus in the narrow-phase is to carefully check these pairs for collision.

A brute-force implementation of the narrow-phase takes every pair of the PCS list $P(A, B)$, and will test all primitives of object O_A's geometry again all primitives of object O_B's geometry. This brute-force approach will be in $\mathcal{O}(n \cdot k)$, while n is the number of primitives of object O_A and k is the number of primitives of O_B. Due to the fact, that in most scenes the complexity of objects are much higher than the number of objects within a scene, a quadratic complexity for the narrow-phase is not a feasibility in nearly all situations. That is the reason why so many different approaches exist to speed up the narrow-phase collision detection.

It must be noted that in the case of soft collision detection *all* kind of collisions—collisions between objects and self-collisions—must be considered and therefore, not only objects which are in the PCS are going further investigations. In the narrow-phase all objects have to be checked for self-collisions, which can be done natively by checking a primitive of the object against all other primitives of the same object.

The narrow-phase part of the collision detection process can be subdivided into three challenges:

1. Remove all definitively non-colliding pairs and find which sub-objects are really intersecting.
2. Determine all needed proximity/contact information, e. g., exact contact points where objects are touching (inter-penetrating), surface normal at that contact points, penetrating distance or penetration volume.
3. Detect resting or persistent contacts, i. e., equivalent contact points from previous simulation steps.

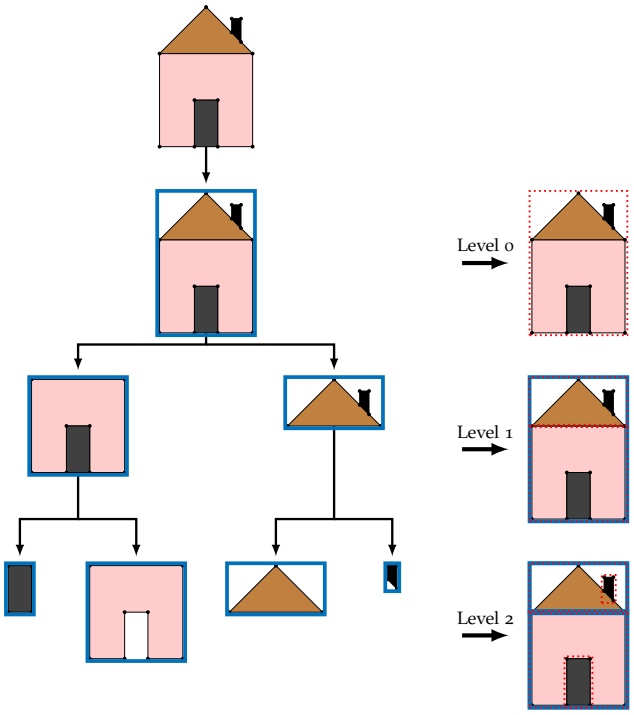


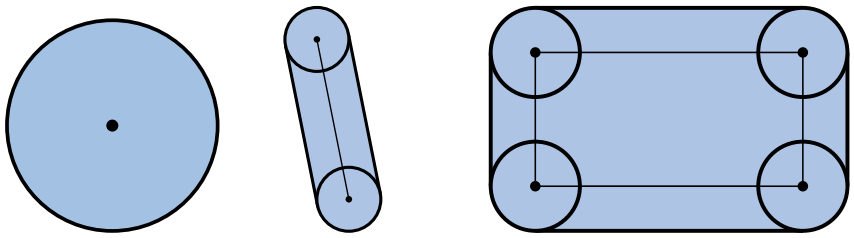
Figure 2.6: An example of a simple BVH using AABBs as Bounding Volume. On the left side a geometric object is subdivided recursively into subsets of their primitives or groups of primitives. The right part of the Figure depicts all BVs for corresponding level of the hierarchy. A higher number of level leads to a finer structure of BVs.

For rigid body collision detection the common data structure, used to reduce the number of intersection tests, are BVHs (see Figure 2.6). To generate a BVH, a geometric object is recursively divided into groups of primitives encapsulated by a BV, which are subdivided again. This procedure is repeated until the BV bounds a predefined number of primitives. Every object can be used as a BV, but most of them will not serve as an effective BV. There are some requirements to be an effective BV:

- Tight fitting to well-approximate the bounded object
- Efficient intersection/overlap test

- Efficient creation
- Efficient updating (transformation)
- Memory efficient
- Well-suited for hierarchy construction

However, most of these requirements are contrary. Some BV approximate the shape of an object better but the intersection test is much more expensive. Furthermore, in the case of deformable objects a BV which approximate the underlying geometry perfectly at the beginning of a simulation could perform poorly at the end, because of an intensively deformation. Commonly used BVs are AABB in a global reference [Hah88; LA01], or in a local reference of the model [Ber98], OBB [GLM96], spheres [Hub96; PG95; Rit90], k-Discrete Oriented Polytopes (k-DOP)[2] [Klo+98; Klo98; Zac98a], Fixed-Direction Hulls (FDHs) [KZ97], or convex hull (few examples of BVs are shown in Figure 2.5). Other often used BVs are sphere-swept volumes. Therefore, a sphere slides along the border of an object, e. g., point, line, rectangle and so on. Figure 2.7 depicts some simple sphere-swept volumes, however every convex object can be used as swept volume but most more complex objects have a more expensive intersection test.



(a) Sphere-Swept Point (SSP) (b) Sphere-Swept Line (SSL) (c) Sphere-Swept Rectangle (SSR)

Figure 2.7: Some simple examples of sphere-swept volumes.

Because of the fact that the best BV is very dependent on the underlying problem further types have been suggested as BV. For example, cones [Ebe02; Hel97], cylinders [Ebe00; Hel97; Sch+00], spherical shells [Kri+98], Quantized Orientation Slabs with Primary Orientations (QuOSPO) [He99], ellipsoids [CWK03; RB92; Wan+04; WWK01], and zonotopes [GNZ03].

But there is more required than just a BV to generate a *good* BVH. Therefore, numerous parameters must take into account and choose wisely.

- Type of Bounding Volume: AABB, OBB, sphere, ...

2 k-DOPs are polytopes based on the AABB.

- Type of tree (binary, 4-ary, ...)
- Insertion/bottom-up/top-down construction method
- Heuristic to subdivide/group object's primitives or BVs
- How many primitives in each leaf of the BVH

We already introduced the most commonly used BV in the Section 2.5. Another parameter is the maximum number of children per node, typical is a binary, but also quad- or n-ary trees are possible. The next parameter is the technique used to construct the hierarchical tree. There are three mainly used construction techniques: insertion methods [Bec+90; GS87], bottom-up methods [Bar+96], and top-down methods [TR98]. During the hierarchy construction process a heuristic is needed to decide if a BV is subdivide—top-down construction approach—or if BVs or on the first level object's primitives are grouped—bottom-up approach. The last parameter specifies the number of primitives stored in a leaf of the BVH and therefore, a maximum number of elements a leaf BV encloses. Over time, some different BVHs have been developed, for example Quadtrees [Sam84], Boxtree [Bar+96; Zac95] and k-DOP trees [Klo+98; Zac98b].

Weghorst, Hooper, and Greenberg [WHG84] introduced a cost function to analyze the computation cost for hierarchical structures of BVs in the context of ray tracing. This cost function was used by Gottschalk, Lin, and Manocha [GLM96] also, but within the context of collision detection. The total cost to check two hierarchies, which approximate two complex models, for intersection was quantified by Gottschalk, Lin, and Manocha [GLM96] as:

$$T_I = N_v \cdot C_v + N_p \cdot C_p \tag{2.1}$$

with

T_I	Total cost function for intersection test
N_v	Number of BV overlapping tests
C_v	Cost for average overlap test
N_p	Number of primitive pairs tested for interference
C_p	Cost for average primitive pair test

The Eq. (2.1) is well-suited for measuring of the hierarchy traversal cost associated with performing a single intersection detection check between two complex models. In the case of a deformable model the shape of this object can change and therefore, the BVH can become invalid. To ensure that the BVH is valid we have to maintain an update step. Taking this into account will extend the previously introduced cost function T_I (see Eq. (2.1)) to:

$$T_{UI} = N_u \cdot C_u + N_v \cdot C_v + N_p \cdot C_p \tag{2.2}$$

where N_v, C_v, N_p, and C_p are defined as in Eq. (2.1) and

T_{UI}	Total cost function for hierarchy updates and intersection tests
N_u	Number of BV updates
C_u	Cost for average BV update

Optimizing a BVH is in most cases problem specific; furthermore, lowering one factor often raises others. Another problem using BVH is the update procedure to ensure the BVH is still valid. This process depends on the degree of deformation, like we show in Eq. (2.2). In the case of a high degree of deformation we need to update all the BVs. Therefore, the update cost function will be $N_u \cdot C_u$, with N_u number of all BVs in the scene, for the update process only. It should be noted, that a BVH for deformable models has usually more BVs than primitives in the scene. Kopta, Ize, Spjut, Brunvand, Davis, and Kensler [Kop+12] use tree rotations to efficiently refitting the BVH. In static scenes this technique can be used to off-line improving the quality of the BVH. This approach is fast, has minor update time and will not degenerate the tree. But their results showed that in some cases the quality of the tree got worse. That problem will increase in highly dynamic scenes. It should be mentioned that hierarchies for deformable collision detection are in most cases based on AABB [Ber98; Smi+95] or k-DOP tree structures because of their faster update procedure, which is necessary for deformable models.

The attentive reader will already have noticed that nearly all BVs are convex. There are two main issues using convex BVs in the context of collision detection. The first one is the existence of separating plane if two convex models do not intersect. The other reason is that the minimal distance between two convex models is a local minimum. Therefore, a simple *gradient ascent algorithm* (often referred to as *hill-climbing algorithm*) will find the minimum distance [KT06]. Taking this in mind it is possible to generate efficient collision detection algorithm for convex objects. The first group of collision detection approaches for convex objects we discuss are *closest-features* based algorithms. Closest-features are vertices, edges, or faces from each object, which are next to the *real* closest point between two objects. The *Lin-Canny algorithm* [LC91] introduced in 1991 is the first closest-features based collision detection approach ever. Till today this algorithm is one of the most well-known closest-features based algorithm. Mirtich [Mir98] presented *Voronoi-Clip*, or *V-Clip*, a collision detection approach using the boundary representation of a polyhedral model. This approach tracks the closest-features between convex models. Concave objects are also supported; they are subdivided into convex subparts. Another very efficient method to determine

the minimum distance, and therefore, the intersection between two convex objects is the *Gilbert-Johnson-Keerthi (GJK) distance algorithm*, commonly referred to as the *GJK algorithm* [GJK88]. This approach uses two sets of vertices and constructs a convex hull for each input set. Now the algorithm computes the minimum Euclidean distance, beside the closest-points between two convex hulls. Gilbert and Foo [GF90] presented an extended version of the standard GJK approach to support arbitrary convex point sets, e. g., point clouds. The GJK algorithm does not work on the two input point sets directly, for the distance computation process it uses the Minkowski difference between both sets. Another convexity based collision detection algorithm is *Chung–Wang separating-vector algorithm* [CW96]. This approach efficient computes a separating axis, which in the case of two convex objects does not intersect with the objects. All convexity-based algorithms are not especially well-suited for deformable models, because objects which are convex at the beginning of a simulation, can become concave after some simulation steps. One possibility is to subdivide the concave object into convex parts but doing this every frame will lead to a high computation time. Another important fact is that all these approaches do not support self-intersections and therefore, every object has to be further investigated by another algorithm.

2.7 SOPHISTICATED NARROW-PHASE COLLISION DETECTION: SWEEP-AND-PRUNE

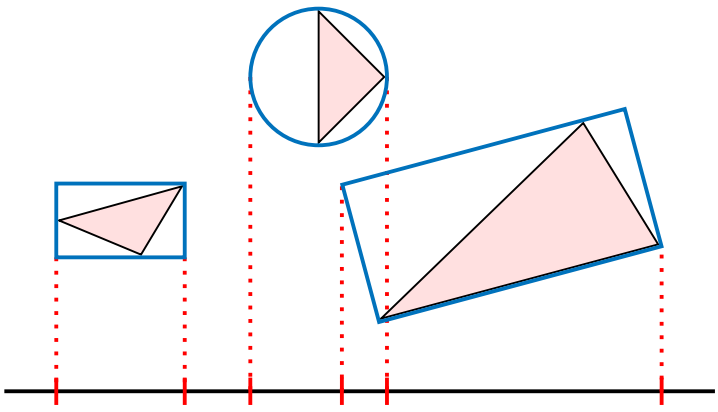


Figure 2.8: This example depicts the projection process of different types of BVs (AABB, sphere and OBB) onto an axis.

Like we mentioned in the Section 2.5 the Sweep-and-Prune (SaP) approach is a topological method which is based—in the case of a broad-phase approach—on the relative position of objects to each other. The narrow-phase focuses primarily on the exact collision detection on the level of primitives. Therefore, the relative position of primitives to each other is more important. To achieve this, the BVs

enclose the primitives and the exact collision detection process can be described as a problem of finding all overlapping pairs of BVs.

If we project the boundary of all BVs onto all axes (see Figure 2.8), this problem can be reduced to searching all overlapping one-dimensional intervals along all axes. Figure 2.8 depicts that using AABBs as BV is the most suitable variant. For rigid body collision detection different data structures have been investigated to solve this problem. These data structures are segment trees and interval trees [Ove88; SW82], and R*-trees [Bec+90; BS09]. R*-trees are well-suited for querying n-dimensional problems, but it is not clear how these data structures can be used for highly dynamic simulations, where nearly all BVs move and/or change their size.

For the $\mathbb{R}^3$ case algorithms exist with a time complexity of $\mathcal{O}(n \cdot \log n + k)$, where k is the number of pairwise primitive overlap tests. Preparata and Shamos [PS85] showed that a general d-dimensional Bounding Volume intersection test can be solved in $\mathcal{O}(n \cdot \log^{d-2} n + k)$. But it should be noted that these algorithms are *output-sensitive*. Therefore, in worst-case scenario exact collision detection approaches, which have to find and test *all* intersecting pairs, are still in $\mathcal{O}(n^2)$.

We start with the simplest case, 1 D, and abstracting this method to a more complex 2 D or d D rectangular range searching problem. Let us assume that there is a scene containing a number of primitives n. For each primitive p_i, $i = 0, \ldots, n-1$, we compute the corresponding AABB, hereinafter referred to as BB_i. The main idea is now to project all BB_i onto an axis. Each BB_i spans an interval $[S_i, E_i] \subset \mathbb{R}^1$ for each primitive p_i on the axis. The problem is to determine for all pairs i and j, whether the intervals $[S_i, E_i]$ and $[S_j, E_j]$ overlap or not.

This problem can be solved with the Sweep-and-Prune algorithm. Sorting all intervals along an axis provides information about possible colliding BVs because, two BVs i and j collide, iff one of the four cases $[S_i, S_j, E_j, E_i]$, $[S_j, S_i, E_i, E_j]$, $[S_i, S_j, E_i, E_j]$, or $[S_j, S_i, E_j, E_i]$ occurs (see Figure 2.9). Detecting all intersecting intervals is often referred to as *sweep* step.

During the sweep process an additional list of *active* intervals, which is initially empty, is maintained. A primitive p_i is added to the active interval list at the moment their S_i value is passed the sweep process. A primitive p_i will be removed from the active interval list when their E_i value is encountered. When the sweep process adds a new primitive p_i to the active list, the interval of p_i overlaps with *all* primitives in the current active interval list. This procedure determines all pairs of overlapping intervals that have to be further investigated. The time need to sort the n primitives is in $\mathcal{O}(n \cdot \log n)$. The sweep process is in $\mathcal{O}(n)$ and the output process is in $\mathcal{O}(k)$, where k is the number of overlapping BV intervals. This results in an overall complexity in $\mathcal{O}(n \cdot \log n + k)$, which is the optimal running time to solve this problem.

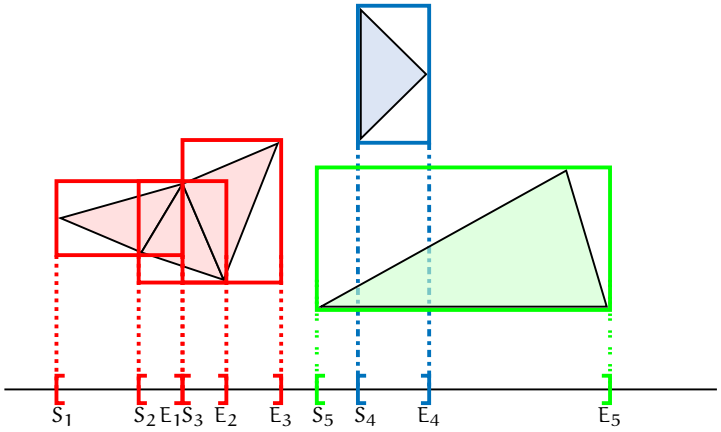


Figure 2.9: The example scene consists of three different models colored in blue, red and green. For all primitives in the scene, their BV is projected onto an axis. For every BV the projected AABB interval is represented by an opening (S_i) and a closing (E_i) bracket, with i is the ID of the corresponding BV.

Theorem 2.7.1 (1 D Range Searching) *A fixed number of points n can be stored in a balanced binary tree. This results in a space complexity of $\mathcal{O}(n)$ and a construction time in $\mathcal{O}(n \cdot \log n)$. To report all points in a query range the algorithm has a complexity of $\mathcal{O}(n \cdot \log n + k)$, with k is the number of reported points [Bar92; Ber+08; Ove88].*

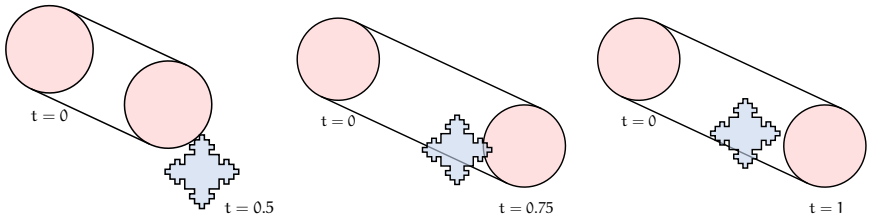
Because of the fact that in dynamic scenes the movement and deformation of objects is not very heavily from frame to frame sorting the list from the beginning every frame is not necessary. Therefore, Baraff [Bar92] advises to use *insertion sort* [SW11] to permute the *nearly* sorted list into a sorted one. Insertion sort is in $\mathcal{O}(n+s)$, where s is the number of swaps to order the input list. Zachmann [Zac00] used *Bubblesort* [SW11] to utilized temporal coherence. Assuming coherence, this approach will be in $\mathcal{O}(n+s)$ for the 1D BV problem. Therefore, the introduced algorithm is efficient and makes an extreme simplify exploitation of coherences within a scene available.

To extend this approach to more than one dimension, for each dimension in d D an independent interval for all primitives p_i is needed, $[S_{v_i}, E_{v_i}]$, $v = 1, \ldots, d$. Now all intervals are sorted like in the 1D case, and d active interval lists are maintained. If two BV intervals overlap in all d dimensions, both primitives can potentially intersect and an exact intersection test has to be performed. Therefore, this approach is simple to extend to more than just one dimension.

For most 3 D scenes, sorting on just two axes remove most of the false-positives cases. Furthermore, omitting one dimension will reduce the memory overhead.

2.8 CONTINUOUS COLLISION DETECTION

Non-continuous collision detection discretized time and therefore, objects position is changing discontinuous from one time step to the next one. Choosing a good time step t is important for many simulations. Figure 2.10 depicts the movement of two objects for three different time step values. A non-continuous collision detection approach will only detect a correct collision for $t = 0.5$ (see Figure 2.10a). For $t = 0.75$ the collision will be recognized too late and for $t = 1$ the collision will be missed completely (see Figure 2.10b and 2.10c). This can result in visual artifacts in physical-based simulations, e. g., a collision between two objects is detected by the collision detection system and they bounce away from each other, although a collision cannot be detected visually. Another problem is the so-called *tunneling effect*, which occurs if objects move too fast, or the time step between two collision queries is too large, the object could pass through another one or collide with the backside and move to a wrong direction. To avoid this kind of errors, the exact time of collision between the objects needs to be determined [Cou05]. To solve this problem of *continuous collision detection*, or often-called *dynamic collision detection*, some different techniques have been developed.



(a) Collision recognized correctly (b) Collision recognized too late (c) Collision missed

Figure 2.10: Simulation states for different step size t.

The simplest technique is to *simulate* continuous collision detection. Therefore, we use a smaller time step size for the collision detection process than used for the visual system. Boolean rigid body collision detection algorithms only need a few milliseconds, depending on the object's configuration, whereas visual interactive applications usually need an update rate about 30 frames per second, which means that around 30 milliseconds passes between two static collision detection tests. This technique is called *pseudo-continuous collision detection* and makes it possible to perform more static collision detection checks between two steps of the visual simulation [HKM96].

Conservative advancement, presented by Brian Mirtich [Lin93; Mir96; MC95], is another simple technique for continuous collision detection. Therefore, this approach computes a lower bound of the first time of contact—or, to be more precise, the times of discontinuities—between two convex objects. The non-penetration constrain is guar-

anteed by taking smaller time steps if the simulator tends to creep up to each discontinuity [Miro0]. Mirtich [Miro0] noted that conservative advancement become intractable with many bodies within a scene. Coumans [Cou05] presented a specialized form of conservative advancement. Bergen [Ber04] used a deformed Minkowski sum in combination with ray casting.

A further possibility for continuous collision detection is the use of swept volumes. Therefore, the swept volume encloses the BV of an object or primitive at time step t_0 and the BV of this object or primitive at time step t_1. Eckstein and Schömer [ES99] showed that using swept volumes could be done efficient for different BV types like AABB, OBB and more. Redon, Kheddar, and Coquillart [RKC02] use Interval Arithmetic (IA) to compute inter-penetration times between object's features (vertices, edges and faces). For the continuous collision detection check they derive a continuous overlap test for moving OBBs. Weller and Zachmann [WZ06] presented an event-based approach where they use a new acceleration data structure, the *kinetic separation list*. The main idea is to transform the continuous problem into a discrete one. This step reduces the number of BV checks significantly. Their approach is fifty times faster than the classical swept volume algorithm.

However, many more approaches that focus on accuracy instead of the running time have been developed also. Snyder, Woodbury, Fleischer, Currin, and Barr [Sny+93] use IA and interval Newton methods to speedup their approach. This algorithm is very well-suited for computer graphics animations but is not fast enough to perform real-time interactions [Sny95]. Canny [Can86] uses quaternions instead of Euler angles for the objects trajectories. A down side of this approach is the high computational complexity, which does not run in real-time for large models. Using screwing's [KR03] to parameterize object's trajectories is another way to move models. Redon, Kheddar, and Coquillart [RKC00] combine screw motion with IA. Von Herzen, Barr, and Zatz [VBZ90] provided a continuous collision detection approach for time-dependent parametric surfaces. Therefore, they use Lipschitz bounds in combination with binary subdivision to find the first time of impact.

2.9 PARALLEL COLLISION DETECTION

As we mentioned in the INTRODUCTION and explain in detail in Section 3, parallel computing has become more and more important. Therefore, many parallel collision detection approaches have been devised in the last decades. Zachmann [Zac01] pointed out several possibilities to parallelize the collision detection process and they are pipelining, concurrency, coarse- and fine-grain parallelization. In this

Section we present some multi-core CPU, hybrid CPU/GPU and full GPU-based approaches.

2.9.1 *CPU*

Huagen, Zhaowei, and Qunsheng [HZQ01] build a hybrid Bounding Volume Hierarch in parallel to speedup the collision detection process. Their approach focuses on MIMD systems and furthermore, they support multi-threading also. This provides the possibility to run on both single processors as well as on multi-processors. Lario, Garcia, Prieto, and Tirado [Lar+01] analyzed an Open Multi-Processing (OpenMP)-based parallelization of a virtual cloth simulator. Therefore, they use OpenMP to parallelize the Multilevel Polak-Ribiere (MPR) algorithm [PR69]. Furthermore, they showed that their solution can efficiently using up to 16 processors. Thomaszewski and Blochinger [TB07] presented an approach for distributed memory architectures. Therefore, they use a data parallel framework (Single-Program, Multiple-Data (SPMD)) to seamless integrates the parallel collision handling phase into the physical modeling phase. Selle, Su, Irving, and Fedkiw [Sel+09] use a recursive median split to distribute the particles of the cloth mesh to different processors. They synchronize the BV position data to every processor. Therefore, they have a full hierarchy for all primitives, inclusive points, segments, and triangles, which they use in the traversal step on every processor. Furthermore, their approach uses a Gauss-Seidel ordering step to ensure each pair process seeing the newest data. Thomaszewski, Pabst, and Blochinger [TPB08] analyze previous simulation steps to subdivide the problem into equally computational parts. Furthermore, they use key techniques for parallel physically-based simulations to eliminate the two major bottlenecks of the simulation—the solution of the linear system and the collision handling process. Tang, Manocha, and Tong [TMT09] presented a multi-core approach for continuous collision detection. They parallelize incremental hierarchical computation among multiple cores of GPUs. This approach scales very well with the number of cores. Kim, Heo, and Yoon [KHY09] use a feature-based BVH to speedup the collision detection process. They introduce a novel task decomposition method for BVH-based approaches. Furthermore, they use a dynamic task assignment method for the distribution process, to achieve better utilization of the available computation power. Before a simulation step is performed Hermann, Raffin, Faure, et al. [H+09] subdivide a task dependency graph to disturb tasks between processors. Therefore, they extract tasks from the sequential algorithm and generate from this input a data-flow graph. To avoid expensive graph partitioning processes, they limit the intensity of the changes between two frames. Tang, Manocha, and Tong [TMT10] use fine-grained front-based decomposition to spread

the computation process among 8 core and 16 core PCs. To detect all inter-object collisions as well as self-collision they use an adaptive rebuilding process. Their timings show $6.4\times - 7.7\times$ speedups in the overall running times on 8 cores, and $10.1\times - 13\times$ speedups on 16 cores.

2.9.2 *Hybrid CPU — GPU*

Kim, Heo, Huh, Kim, and Yoon [Kim+09] presented an approach using a BVH. The BVH traversal and culling steps are performed on the CPU, while elementary tests are completely performed on the GPU. They introduce a lock-free parallel algorithm for main loop of the collision detection by using a novel task decomposition method. Pabst, Koch, and Straßer [PKS10] use a master-slave system, where a master-thread on the CPU delegates the work to slave-threads. Every single slave-thread performs the computation at their assigned GPU. Furthermore, all slave-threads are running asynchronous, which enables overlapping master/slave CPU/GPU computations.

2.9.3 *GPU*

Wong and Baciu [WB05] showed that using GPU as computing device is much faster instead of using the CPU. Therefore, they subdivide the collision detection process in a set of fine-grained tasks. Govindaraju, Lin, and Manocha [GLM05] run visibility queries on the GPU to reduce primitive intersection tests, which are not in close proximity. Greß, Guthe, and Klein [GGK06] presented a collision detection approach for deformable parameterized surfaces. They represent the individual parameterized surfaces by stenciled geometry images, which they use to create a BVH. This BVH serve as a basis for the optimized collision detection approach. Lauterbach, Mo, and Manocha [LMM10] use data parallelism to perform fast hierarchy construction, updating, and traversal steps. Therefore, they use a tight-fitting BV, like OBB and SSR. Furthermore, using tight-fitting BVHs are extremely advantageous for GPUs. Tang, Manocha, Lin, and Tong [Tan+11] use a hierarchical culling in combination with the generation of different streams to reduce the computation. Furthermore, using a deferred front tracking method reduces the memory overhead. Pan and Manocha [PM12] presented an approach, which use a clustering scheme and collision-packet traversal to perform an efficient collision detection process. They use a hierarchical traversal scheme that performs workload balancing.

2.10 TIME-CRITICAL COLLISION DETECTION

Many simulations focus on running in real-time to be highly interactive, such as virtual prototyping, 3 D video games, and fly simulators. Most applications simulate more or less physically correct behavior. The reason for this is, that humans cannot distinguish between simulations are physically correct and physically plausible (up to some degree) [BHW96]. This opens up the possibility to be less accurate while speedup the collision detection, if real-time collision response is a crucially depending. Therefore, approaches which approximate the object's shape or using probabilities to decide if objects intersect or not are interesting alternatives.

Hubbard [Hub96] approximates the object surface by a hierarchy of spheres. Level 0 is the object's bounding sphere. A deeper level of the hierarchy use unions of successively more spheres, representing the object's surface. For the collision detection process this approach tests each level of the hierarchy for overlap and stops if a critical computation time is reached.

Klein and Zachmann [KZ03b] presented a framework that allows an application to control speed and quality of the collision detection. They use a probability measure to decide if a pair of BVs contains intersecting polygons or not [KZ03a]. Since our focus is on exact collision detection these techniques are not sufficient for our approach.

2.11 RELATED FIELDS

Acceleration data structures for geometric queries are not only restricted to the field of collision detection only. There are many other fields that could benefit from these data structures. Just to name a few, without any claim to completeness: ray tracing (see Section 2.11.1), volume rendering (see Section 2.11.2), occlusion culling [GSF99; GKM93; YG07; Zha+97], view frustum culling [AM00; Cla76], backface culling [Van94; ZH97], object tracking [MZ10], audio rendering [Tsi+07; WS04], robot motion planning [Ber+08] and path planning [Gay+05].

2.11.1 *Excursus: Ray Tracing*

Whitted [Whi80] introduced the basic concept of ray tracing in 1980. With ray tracing it is possible to render photo-realistic images from a virtual scene (see Figure 2.11). Therefore, a light ray is traced backwards from the point of view to all light sources. Should a ray hit an object within the scene, an additional ray is shot to the light sources and moreover. Furthermore, refracted and reflected rays are traced recursively through the scene. This technique offers the opportunity to simulate global illumination effects, if a ray hits a surface three

Figure 2.11: Virtual scene rendered with Blenders new unbiased rendering engine called Cycles. Ray tracing offers many possibilities, like reflections, refractions, and shadows

new types of rays—reflection, refraction, and shadow—can be generated. The general principle is to find all intersections between rays and objects within the scene. The problem of testing for intersection is closely related to collision detection and therefore, the geometric acceleration data structures are very similar.

Kay and Kajiya [KK86] use object's normals to generate a convex hull as BV. Furthermore, they create a hierarchy of this BV to speedup the ray tracing process. Goldsmith and Salmon [GS87] introduced an automatically BVH generation method for use in ray tracing. A hierarchical grid as acceleration data structure for dynamic scenes has been used in [RSH00]. Wald, Slusallek, Benthin, and Wagner [Wal+01] provided a new algorithm, which perfectly uses caches and SIMD instructions. Furthermore, they exploit image and object space coherences. This leads to an implementation, which can render complex scenes at interactive rates. Foley and Sugerman [FS05] presented two kd-tree traversal approaches, which are especially well-suited for GPU implementation. They showed that their implementation is much faster than using a uniform grid as acceleration data structure. Wald, Boulos, and Shirley [WBS07] use for the BVH construction step a variant of the Surface Area Heuristic (SAH). Additionally they do not change the structure of the BVH, thus only the BVs have to be updated from one frame to the next one. Last but not least, a new packet-frustum traversal scheme is to pack rays and trace them together.

Over time and as progress has been made in computer hardware more and more parallel approaches have been developed. The parallel power allows the rendering of dynamic scene in near real-time processing time [Dje+07; Hor+07; Kan+13; Par+10]. Buck, Foley, Horn, Sugerman, Fatahalian, Houston, and Hanrahan [Buc+04] and Carr,

Hall, and Hart [CHH02] showed in experiments that GPU can outperform CPU implementations for ray-triangle intersection tests.

2.11.2 *Excursus: Volumen Rendering*

Figure 2.12: Volume rendering of a human's head with torso (Osirix Manix data set).

The main idea of volume rendering is to render a 2 D image from a 3 D discretely sample data set. Most data sets are acquired by Computed Tomography (CT), Magnetic Resonance Imaging (MRI), or MicroCT scanner. There are some commonly used techniques to generate a 2 D image from a volume data set. One technique is to extract isosurfaces from the data set and rendering them as polygonal meshes or by rendering the 3 D data set directly as voxels. To generate an isosurfaces from a 3 D set the marching cube algorithm [LC87] is typically used. For direct rendering a transfer function is needed to map opacity and color to the output pixels. It should be noted that direct rendering is a much more computational intensive task.

Volume ray casting is a technique to generate high quality images from data sets. Therefore, a huge amount of rays are traced through the data set or the generated isosurface. This is again closely retailed to the process of collision detection—intersection test between a ray and a surface or a voxel based representation—and therefore, the geometric acceleration data structures are very similar.

The approach presented by Laur and Hanrahan [LH91] based on a pyramidal volume representation. Furthermore, they fit an octree to the pyramid to improve the performance of the volume rendering process. Lin and Ching [LC96] use an octree to efficiently determine all cells the isosurface intersects. Neubauer, Mroz, Hauser, and Wegenkittl [Neu+02] presented a new technique for fast perspective volume visualization. Therefore, the volume is divided into cubic subvolumes. Which are stored in a hierarchy. For efficiently traverse through the 3 D data set Parker, Shirley, Livnat, Hansen, and Sloan [Par+98] use a multi-level spatial hierarchy. Leven, Corso, Cohen,

and Kumar [Lev+02] use an octree in combination with a tetrahedralization for volume rendering process. To accelerate the ray tracing for volume rendering Jamriška [Jam10] represents the distance fields by sparse block grid data structure. Knoll, Thelen, Wald, Hansen, Hagen, and Papka [Kno+11] presented an efficient method for volume rendering by raycasting using the CPU. They deploy coherent packet traversal of an implicit BVH to exploit empty or homogeneous space. Fraedrich, Auer, and Westermann [FAW10] presented an approach for high quality volume rendering of SPH data using a novel view-space discretization of the simulation domain. Their approach is based on recent work on GPU-based particle voxelization for the efficient resampling of particles into uniform grids. Hassan, Fluke, and Barnes [HFB12] use a kd-tree to speed up the the volume rendering process over a GPU-cluster. Kroes, Post, and Botha [KPB12] presented a GPU-based approach with support for physically based lighting, which improve optical realism.

3

A BIREF INTRODUCTION INTO MASSIVELY PARALLEL COMPUTING

> "Concurrency has long been touted as the "next big thing" and "the way of the future," but for the past 30 years, mainstream software development has been able to ignore it. Our parallel future has finally arrived: new machines will be parallel machines, and this will require major changes in the way we develop software."
>
> Herb Sutter and James Larus, Microsoft, 2005 [SL05, Section 1]

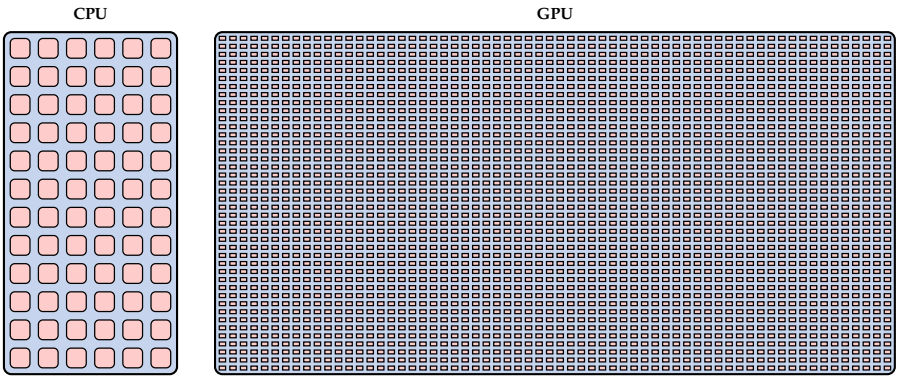


Figure 3.1: This Figure compares the amount of threads available on a current CPU and GPU. The Intel® Core™ i7-4960X CPU has 72 threads in total, 6 cores with 12 threads for each core. The NVIDIA GeForce GTX TITAN has 2688 CUDA cores, which are equivalent to the CPU threads.

As already mentioned in the INTRODUCTION (Section 1), the current trend in computer architecture focuses on multi-core CPUs and many-core GPUs (see Figure 3.1). A multi-core CPU focuses on fast serial processing, for which the cores are latency-optimized. In contrast to this, many-core GPU focuses on scalable parallel processing, for which the cores are throughput-optimized. Since the performance of the GPU increases faster than the power of the CPU in the last years, more and more investigations to perform General Purpose Computation on GPUs (GPGPU) have been done (see Figure 1.2). In 2007 NVIDIA introduced CUDA, which opens the general public the power of the GPU-computing and to perform GPGPU. NVIDIA indicates that their exits 1000's of GPU-accelerated applications using CUDA and 1000's of published research papers using CUDA. Furthermore, there are many other applications using Open Computing Language (OpenCL) to use the GPU as a coprocessor. All these ele-

ments clearly show that currently the main focus lies on GPU-based approaches.

Therefore, a huge amount of investigation has been done to fully exploit the GPU hardware, thus many applications and algorithms has been developed. Thus, we can mention only a few approaches performing GPGPU at this point and these are: simulate the dynamics of clouds on graphics hardware [Har+03], implement different ray tracing approaches [CHH02; Par+10; Pur+02], perform Fast Fourier Transform (FFT) [MA03], implement linear algebra operators [KW03], solve sparse matrices [MA03], solve the Navier-Strokes equations [Har05], sort huge input data [GZ06], compute a space filling sphere packing for arbitrary objects [WZ10], Batch Neural Gas (BNG)-based hierarchy construction [Wel+14], and many more.

3.1 THE GRAPHICS HARDWARE

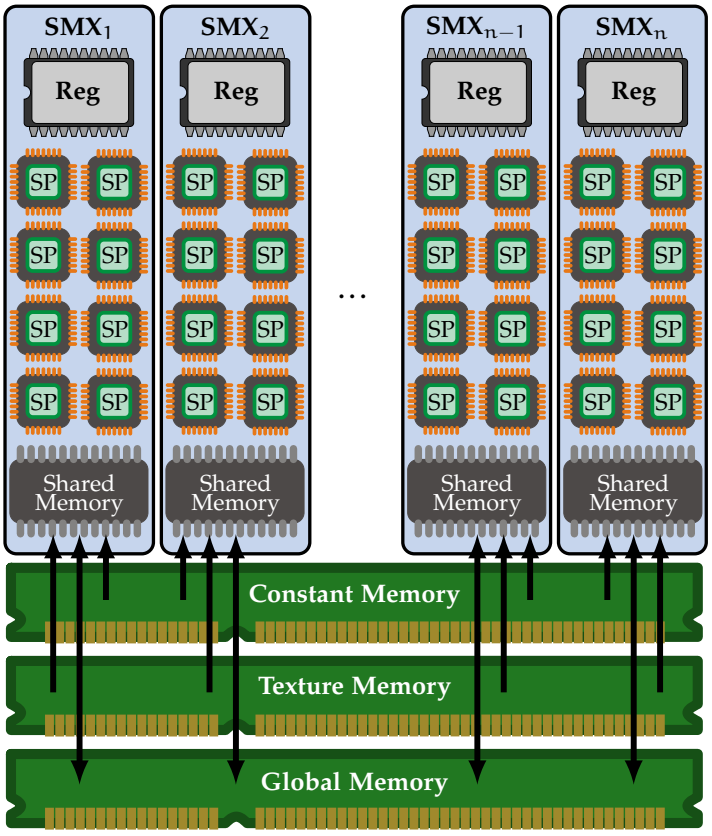


Figure 3.2: With the Kepler architecture NVIDIA introduced the new Next Generation Streaming Multiprocessor (SMX)-design. With the new SMX-design NVIDIA increases the number of processing units up to 192 CUDA cores per SMX.

The Figure 3.2 provides a simplified presentation of a GPU. In this Section we use notations introduced by NVIDIA, for a conversion between CUDA and OpenCL notations we want to refer to Table A.1.

The massively parallel GK110 chip of NVIDIA GeForce GTX TITAN has 2688 Streaming Processors (SPs) (14 SMXs, each with 192 SPs). Every CUDA device has several different memory types: Global, texture, constant, shared and register memory. Every memory type has both advantages and drawbacks. Using a wrong memory type within the application can drastically slow down the computation process. In Table 3.1 we give a short overview of access times for different memory types. Registers are the fastest way to access data

#CLOCK CYCLES	MEMORY TYPE
0	Register memory
1–2	Shared memory
4 or 64	Constant memory (same or diff. location)
100's	Global GDDR5 memory

Table 3.1: Different memory types with the corresponding access time. The constant memory has two different access times, on the one hand, if all threads access the same memory location, and on the other hand if all threads access different memory locations.

on GPU but there are only 64 k 32 bit register per SMX on GPUs with CUDA Capability between 3.5 and 5.0. Furthermore, registers are only accessible within one thread. The size of shared memory depends on the GPU version; the NVIDIA GeForce GTX TITAN has a size of 64 KB per SMX of shared memory. Threads within the same SMX can only access this memory. Consequently, no information between different SMXs can be shared using this kind of memory. All these three memory types are only existing as long as the kernel is running. That means it is not possible to store computation results within these memory and use the results in another kernel call. The constant and global memories hold their values as long as the application is running. The size of the constant memory is 64 KB and therefore, only suitability for a few frequently used read-only data. The constant memory can be access from any thread running at the same time. The global memory is the slowest, but also the memory with most space on the GPU—NVIDIA GeForce GTX TITAN has a total of 6144 MB. Any thread running on the GPU has read and write access to this memory. To get best performance this memory should be access in a coalesced way (see Figure 3.3) [Nvi14].

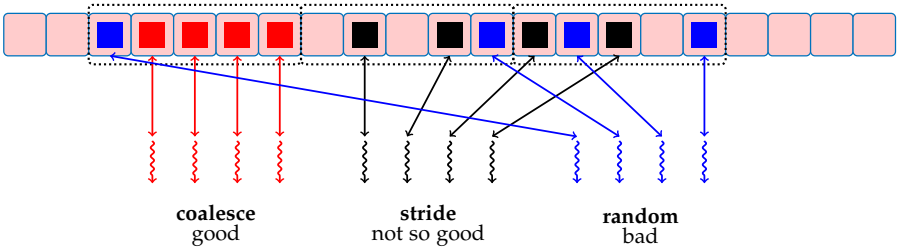


Figure 3.3: Examples of memory access pattern.

3.2 PERFORMANCE OF PARALLEL COMPUTING

In this Section we want to show some parallel techniques and methods to compare the performance of parallel approaches and systems. A multiprocessor consists of n_p processors, with $n_p \in \mathbb{N}$. The *parallel fraction* value of a program is given by $\mathcal{F} \in [0, 1]$. For the rest of this Section we use the following parallel performance notations, which are commonly used [KP12]:

$$\begin{aligned} S_p &= \frac{T_1}{T_p} \quad — \quad \text{speedup value} \\ E_p &= \frac{S_p}{n_p} \quad — \quad \text{efficiency value} \end{aligned} \tag{3.1}$$

where T_1 is the serial computation time of a program, T_p is the parallel wall clock time for executing the program. In parallel computing two laws are commonly used to predict and estimate the parallel performance of a program: *Amdahl's Law* and *Gustafson's Law*. In the following we want to take a closer look at these two laws.

3.2.1 *Amdahl's Law*

> "For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit co-operative solution ... The nature of this overhead (in parallelism) appears to be sequential so that it is unlikely to be amenable to parallel processing techniques. Overhead alone would then place an upper limit on throughput of five to seven times the sequential processing rate, even if the housekeeping were done in a separate processor ... At any point in time it is difficult to foresee how the previous bottlenecks in a sequential computer will be effectively overcome."
>
> Gene M. Amdahl, 1967 [Amd67, p. 483]

Amdahl's Law uses the parallel fraction value $\mathcal{F}$ and the number of processors n_p to estimate the maximal expected improvement, while

parallelizing parts of a program. This leads to the following definition:

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{T_1 \cdot \frac{\mathcal{F}}{n_p} + T_1 \cdot (1 - \mathcal{F})} \leq \frac{1}{1 - \mathcal{F}} \tag{3.2}$$

and an estimation of the maximal expected improvement:

$$\max S_p = \frac{1}{1 - \mathcal{F}} \tag{3.3}$$

if the number of processors $n_p \to \infty$. The Eq. (3.3) is referring to as Amdahl's Law. Amdahl's Law tells us that for a fraction value $\mathcal{F} \approx 1.0$, the maximal expected improvement is very small, independently how many processors n_p are available. Figure 3.4 depicts the importance of the parallel fraction value $\mathcal{F}$ on the maximal expected speedup value for the function $\frac{n_p}{1+\mathcal{F}\cdot(n_p-1)}$ [KP12].

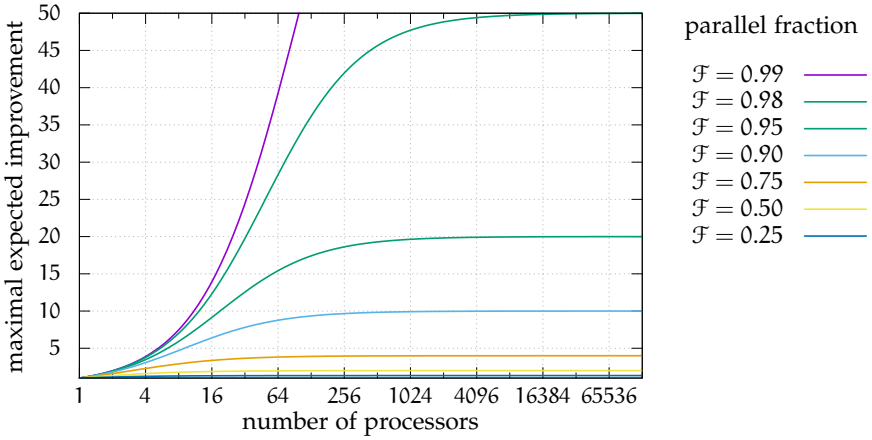


Figure 3.4: The correlation between the parallel fraction value of a program and the maximal expected speedup. If more parts of a program can be parallelized then the maximal expected speedup increase.

The Figure 3.4 shows that programs with a high parallel portion will benefit most from a higher number of processors.

It should be noted that Amdahl's Law is one of the most pessimistic estimation in view of the maximal estimated speedup. Its scale linearly, which means using 10 times the number of processors can speedup a program at most about 10. The Amdahl's Law ignores some facts, which also have positive effects on the speedup factor, e. g., cache per processor, interprocessor communication, and more. For example, suppose you have 10 times the number of processors, and then you have 10 times of cache size also. Therefore, in some cases the whole or large parts of the problem can be loaded into the cache memory and the processors do not need to access the very slow main memory, which will increase the speedup again. Furthermore, Amdahl's Law assumes that the size of a problem is fixed. Increasing the size of a problem, also increase the part, which can be parallelized.

There is also the fact that the greater a problem increase in size; the smaller is the proportion of initializations and synchronization.

Therefore, a more accurate speedup formula is needed, which considered caching, interprocessor communication and some more.

3.2.2 *Gustafson's Law*

For the Gustafson's Law some more notations are needed. The size of a problem is given by n. T_{com} is the time needed for communications between the processors. The Gustafson's serial fraction value is given by:

$$s = \frac{T_{1_s}}{T_{1_s} + \frac{T_{1_p}}{n_p}} \tag{3.4}$$

with

$$T_{1_s} = T_1 \cdot (1 - \mathcal{F}) \qquad \text{and} \qquad T_{1_p} = T_1 \cdot \mathcal{F} \text{ [KP12].}$$

This will result in a speedup factor definition, which takes interprocessor communication into account:

$$S_p = \frac{T_{1_s} + T_{1_p}}{T_{1_s} + \frac{T_{1_p}}{n_p} + T_{com}} \tag{3.5}$$

It should be noted that all provided speedup equations are based on a simple bimodal program structure. This means that a program is executed either serially or in parallel by n_p processors. In most real-world programs several parts of a program have different degree of parallelism. To estimate the speedup for an OpenCL or a CUDA platform the number of processors n_p has to be replaced by the ratio of GPU parallel speed to the CPU speed, given by r_{GC}. This leads to a formula equally to Eq. (3.2), which is the basis of Amdahl's Law.

$$\max S_{r_{GC}} = \frac{1}{\frac{\mathcal{F}}{r_{GC}} + (1 - \mathcal{F})} \tag{3.6}$$

Exploiting Gustafson's serial fraction value Eq. (3.4) leads directly to the Gustafson's Law:

$$\begin{aligned} S_p &= n_p + (1 - n_p) \cdot s \\ \lim_{s \to 0} S_p &= n_p \end{aligned} \tag{3.7}$$

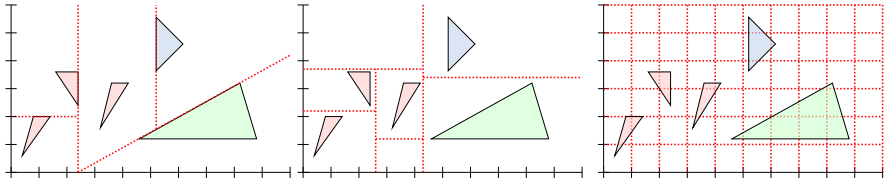
If the sequential execution time is much smaller than the parallel execution time $s \approx 0$—which is true for a larger problem size—the estimated speedup value is n_p. For a more complex program structure than the bimodal we want to refer to a more general approach [Don+98].

3.2.3 *Conclusion*

Both laws, Amdahl's Law and Gustafson's Law, are important to estimate the improvement of a problem using a massively parallel architecture. Amdahl's Law is very pessimistic but it provides a value of an expected improvement, which should be achieved at least. Gustafson's Law, which takes the synchronization and cache usage into account, is a more optimistic law. However, this law expects a very huge input data set, which not hold for all real-world problems. Consequently, both laws should be taken into account to obtain a good maximal expected improvement. Furthermore, the laws show that a good parallelization of a problem is very important to maximize the speedup of a procedure.

4

SCENE SUBDIVISION FOR COLLISION DETECTION



(a) BSP-tree (b) kd-tree (c) Uniform grid

Figure 4.1: This example shows three different techniques for space partitioning.

In the Section Broad-Phase Collision Detection we already mentioned some methods for space partitioning, like uniform grids, BSP-trees, kd-trees (see Figure 4.1) and clustering, a technique known from data mining. In the following, we will have a further look at them and show their advantages and disadvantages. Due to the fact that we focus on deformable collision detection, not only the relative position of objects to each other is important, but rather the relative position of the primitives of all objects to each other. Therefore, all following techniques examine in the context of acceleration of primitive intersection tests. Furthermore, we developed a new volumetric method, which use BNG for hierarchy construction step, to represent 3 D objects by a sphere packing.

4.1 BSP-TREE

BSP-trees have been invented in the context of 3 D computer graphics by Naylor, Amanatides, and Thibault [NAT90] and Naylor [Nay93]. A BSP-tree stores spatial information about objects in a scene. This information is often used to perform geometrical operations in the field of Computer-Aided Design (CAD), collision detection, ray tracing and many more.

In Algorithm 4.1 we present a short overview on the BSP construction process. Constructing the smallest BSP-tree is Non-Deterministic Polynomial Time (NP)-complete and therefore, nobody does this in practice. A much better approach is to use a heuristic to find approximately-optimal trees. Agarwal, Erickson, and Guibas [AEG98] presented a BSP-tree algorithm, which has a construction time of $\mathcal{O}(n \cdot \log^3 n + k \cdot \log n)$ and an update time of $\mathcal{O}(\log^2 n)$ for 3 D scenes, where k is the number of intersecting primitives.

Algorithm 4.1 BSP-tree construction steps

Input: list L of primitives of all objects
Output: BSP-tree
choose a primitive P from the input list L
create a BSP-tree node N
add P to N's polygon list
remove P from L
for all primitives in the list L **do**
  if primitive is completely in front of node containing P **then**
    add primitive to P_{FRONT}
  end if
  if primitive is completely behind node containing P **then**
    add primitive to P_{BACK}
  end if
  if primitive is intersected by node containing P **then**
    split primitive
    add new primitives to P_{FRONT} and P_{BACK} respectively
  end if
end for
apply this algorithm to all primitives in P_{FRONT} and P_{BACK}

4.1.1 *Advantages and Disadvantages*

Since a BSP-tree uses split planes, which are not axis-aligned, additional information have to be stored, like plane's normal and distance from the origin [AT11]. The prevailing belief has been that the quicker traversal of the BSP-tree also speedup intersection tests. However, subdividing primitives who are intersected by a subdivision plane displays another problem. In this way the number of primitives can be raise drastically. Another problem, which may occur within high dynamic scenes—objects are moving widely and/or change their size—is that updates may degenerate a BSP-tree and this will result in notable performance loss. To prevent this, the BSP-tree has to be completely rebuilt from time to time, which will notably reduce the frame rate at this time of the simulation.

4.2 2 D KD-TREE

A kd-tree is an extended version of a binary search tree and was invented by Bentley [Ben75] in 1975. While a binary search tree is used for 1 D range searching problems (see Section 2.7), a kd-tree is employed when the range searching problem has a higher dimension. In the 1 D case you can recursive split the points into two subsets of equal size; one subset holding values greater than a splitting value and the other subset holds the values smaller than the splitting value. The root of the tree stores the splitting value and the two subsets are stored recursively in two subtrees.

For the 2 D problem each data point has two values, a x- and a y-coordinate. The data set is first split on x-coordinate into two subsets,

next on each subset is split on y-coordinate, then again x-coordinate, and so on. This procedure can be directly adopted on the n D case.

A kd-tree can be constructed recursively, like you can see in Algorithm 4.2. The depth is zero in the first call, because we start at the root of the tree. The depth value d is used to decide the splitting axis. The algorithm will return the root node of the resulting kd-tree.

Algorithm 4.2 2 D kd-tree construction steps

```
Input: list L of primitives of all objects, kd-tree depth d
Output: kd-tree t
  function buildKdTree(L, d)
    if L contains only a single point then
      return leaf holding this point
    else if d is even then
      subdivide L with a vertical line l using a split criterion
      into L1 (left of or on l) and L2 (right of l)
    else
      subdivide L with a horizontal line l using a split criterion
      into L1 (below of or on l) and L2 (above of l)
    end if
    t_left ← buildKdTree(L1, d + 1)
    t_right ← buildKdTree(L2, d + 1)
    create node t holding l
    make t_left left child of t
    make t_right right child of t
    return t
  end function
```

Theorem 4.2.1 *For n data points a kd-tree can be constructed in $\mathcal{O}(n \cdot \log n)$ time and need $\mathcal{O}(n)$ space. The output-sensitive query time for a kd-tree is in $\mathcal{O}(n^{1-\frac{1}{d}} + k)$, where k is the number of reported points and d the dimension [Ber+08; Bra08].*

4.2.1 *Advantages and Disadvantages*

Kd-tree is a quite popular data structure in practical applications and conceptually easy to understand and implement. Furthermore, some highly parallel kd-tree approaches exist, which aim to build up the kd-tree in real-time [SSK07; Zho+08].

Just like all other static data structures, a kd-tree needs to be updated frequently to be able to handle dynamic scenes or deformable objects correctly. This update procedure is necessary to ensure that the data structure is still valid after a simulation step. Therefore, a kd-tree can be build up from scratch for every simulation step or a complex update procedure has to be implemented. However, it must be taken into account that in a highly dynamic scene an update step can lead to a very inefficient kd-tree, which results in a deeper tree and therefore, in a slower query response.

4.3 UNIFORM GRIDS

A uniform grid is another very efficient technique to subdivide the space. Therefore, the space is subdivided into a number of regions, or grid cells, of an equal size. Only if two primitives overlap the same cell they could possibly be in contact. If the size of the grid cell is small, then only primitives who are very close together sharing the same grid cells. The computation process of the grid cell coordinates for a data point x_i is very fast and simple:

$$cellCoor_{x,y,z} = \frac{x_{i_{x,y,z}}}{cellSize} \tag{4.1}$$

Neighbor cells from a given cell coordinate are also trivial to locate.

In view of the performance, the number of cells is most important for all grid based approaches. The number of cells of a grid depends directly on the size of the cells. Therefore, choosing a good cell size is most important.

4.3.1 *Advantages and Disadvantages*

Uniform grids are very easy to implement and will perform very well if a good cell size is used. However, a fixed cell size is a downside for any deformable simulation. At the beginning of a simulation setting $cellSize = value_a$ can perform very well but after some simulation steps the grid could be too fine or too coarse. Updating the cell size at running time leads to many other problems. How to determine the best cell size at running time? What if the size of the primitives varies very widely? All these problems have to be handle with if you use a uniform grid for deformable collision detection.

4.4 CLUSTERING — C-MEANS

Clustering is a commonly used approach for data analysis and interpretation to discover structures in the data [And73; Bez81; Dub87; DHS12; Fuk90; JMF99; KR09; KKP05]. The essential component of any kind of clustering or classification is the concept of dissimilarity (distance) or dual similarity. Owing to this concept it is possible to determine how close together two data points are and, based on this result, move them into the same cluster or in two different clusters. The dissimilarity function $d(a, b)$ between the two data points a and b must comply with the following conditions:

$$\begin{aligned} d(a,b) &\geq 0 && \forall\, a, b \text{ and } a \neq b \\ d(a,a) &= 0 && \forall\, a \\ d(a,b) &= d(b,a) \end{aligned} \tag{4.2}$$

Since the focus of this work is on 3 D object interactions, in this context a method is appropriate for the clustering of points in Euclidean space, which is called *sum-of-squares method*. In order to use the distance as clustering criteria a more restrictive concept is needed. For any data points a, b and c the triangular inequality has to be fulfilled:

$$d(a,b) + d(b,c) \geq d(a,c) \tag{4.3}$$

In the Table 4.1 we present some distance functions. Each function describes a different view of the data points in context of their geometrical properties [Ped05].

Hamming distance	(L_1 norm)	$d(a,b) = \sum_{i=1}^{n} \lvert a_i - b_i \rvert$
Euclidean distance	(L_2 norm)	$d(a,b) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$
Tchebychev distance	(L_∞ norm)	$d(a,b) = \max_{i=1,2,\ldots,n} \lvert a_i - b_i \rvert$

Table 4.1: A set of some common distance functions for clustering.

Subdividing or partitioning n data points into c clusters is in most cases a non-trivial problem. Webb [Web03] gave a formula to determine the number of non-trivial partitions for this problem:

$$\frac{1}{c!} \sum_{i=1}^{c} (-1)^{c-i} \binom{c}{i} i^n \tag{4.4}$$

The Eq. (4.4) shows that the number of all possible partitions increases very fast and therefore, enumerate all possible partitions is in most cases unfeasible. In practice a suboptimal solution works fine in most cases. This solution may be determined by the minimization of a certain objective function. The toughest challenge is to determine a suitable objective function, which describes as exactly as possible the problem. Minimizing this problem provides an extremely meaningful structure in the set of data points. Assuming we want to subdivide a set of n data points in $\mathbb{R}^d$ into c clusters, we compute a sum of dispersion between the data points x_i and a set of prototypes (cluster center points) $v_1, v_2, \ldots, v_c$:

$$Q = \sum_{i=1}^{c} \sum_{k=1}^{n} u_{ik} d(x_k, v_i) \tag{4.5}$$

with $d(x_k, v_i)$ being a given fixed distance function (e.g. Euclidean distance, or any l_p-Norm in general, see Table 4.1) between the data points x_k and v_i, the center point of cluster i. Furthermore, Eq. (4.5) contains a partition matrix $U = [u_{ik}]$, $i = 1, 2, \ldots, c$, $k = 1, 2, \ldots, n$,

which allocates the data points to the clusters. In the simplest case the entries in the partition matrix U are 0 or 1. A data point x_k belongs to cluster c_i when $u_{ik} = 1$. For the simplest case, the binary one, Gordon and Henderson [GH77] specific two requirements, which the partition matrix must fulfill:

A cluster c_i is non-trivial, i. e., a cluster does not include all data points and it includes at least one data point:

$$0 < \sum_{k=1}^{n} u_{ik} < n, \quad i = 1, 2, \ldots, c \tag{4.6}$$

Furthermore, a data point belongs to exactly one cluster c_i:

$$\sum_{i=1}^{c} u_{ik} = 1, \quad k = 1, 2, \ldots, n \tag{4.7}$$

Theorem 4.4.1 (Minimization of Q) *The partitioning process can be described as an optimization problem with constraints:*

$\min Q$ *with respect to prototypes* $v_1, v_2, \ldots, v_c$ *and* U *satisfies the conditions of Eq.* (4.6) *and Eq.* (4.7) *[Pedo5; Webo3].*

Many different approaches have been developed to solve this optimization problem. One of the best-known method, to clustering data this way, is *c-means*, or so-called k-means [DHS12; Mac+67; Webo3]. The c-means clustering algorithm directly tries to minimize the quantization error [BB95]. The time complexity of the c-means clustering algorithm is $\mathcal{O}(n \cdot c \cdot d \cdot r_{iter})$, where n is the number of data points, c the number of clusters, d the number of dimensions and r_{iter} the number of iterations. Bezdek [Bez81] described the basic iterative algorithm as follows:

Algorithm 4.3 c-means

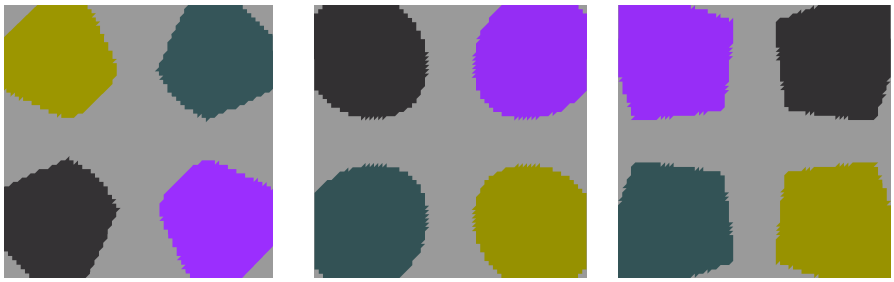
Input: set of data points x_k
Output: partition matrix U
  for all data points x_k **do**
    choose a cluster c_i randomly
  end for
  for all clusters c_i **do**
    compute the centroid
  end for
  repeat
    for all data points x_k **do**
      assign to best cluster ($\min_i d(x_k, v_i)$)
    end for
    for all clusters c_i **do**
      update cluster centroid (use new cluster assignments)
    end for
  until no improvement or until #maxIterations

4.4.1 *Clustering and Classification*

On the basis of the clustering process a meaningful structure in the data set has been detected, which makes it possible to set up a classifier now. A prototype (cluster center point) represents a cluster and therefore, it is the center point of the classifier. A nearest neighbor classification rule specifies that a data point x_k is a member of cluster c_i just when it is the closest to the prototype v_i of cluster c_i:

$$i = \min_j d(x_k, v_j) \tag{4.8}$$

Eq. (4.8) defines a region for each cluster in the feature space. The shape of this region depends on the distance function $d(\dots)$ (see Table 4.1 for some examples of distance functions and Figure 4.2 for the corresponding cluster shapes).



(a) Hamming (L_1 norm) (b) Euclidean (L_2 norm) (c) Tchebychev (L_∞ norm)

Figure 4.2: Examples of distance functions—shape of the clusters for different distance types (see Table 4.1). A plane consisting of 8.5 k triangles is subdivided into 4 clusters. A triangle is assigned to a cluster if their membership value is at least 0.7.

4.4.2 *Advantages and Disadvantages*

In general, researchers from both the optimization and the data handling have heavily investigated the c-means clustering algorithm. Therefore, the c-means algorithm is a very easy to implement and robust, highly efficient and very fast clustering approach. Furthermore, the algorithm runs incrementally and can be parallelized easily. Some disadvantages of the basic c-means algorithm, like being sensitive to outliers, clustering results depends on cluster initialization and performing poorly for non-global clusters—easily gets stuck in local optima—, are sometimes dominated by the advantages, and particularly corrected in new versions of the c-means approach [Wu12].

Pelleg and Moore [PM99] use a kd-tree to improve the performance of the c-means clustering approach. Weber and Zezula [WZ97] showed that bounding trees do not scale well while the dimension increase. Takizawa and Kobayashi [TK06] presented an effective

parallel implementation scheme of c-means clustering. For the subdivision of a large-scale c-means clustering task their approach use a DAC procedure. Other full GPU based implementations have been developed by Farivar, Rebolledo, Chan, and Campbell [Far+08] and Wu, Zhang, and Hsu [WZH09].

4.5 FUZZY CLUSTERING — FUZZY C-MEANS

The fuzzy c-means algorithm is a soft, or fuzzy, version of the well-known c-means clustering algorithm. In the classical c-means clustering algorithm (see Section 4.4) every data point is associated with only the nearest cluster center point. In the fuzzy version of the c-means algorithm, fuzzy c-means, every data point has a membership value u_{ik} in the range of 0 and 1 for every cluster. The algorithm tries to minimize the total error, which is the sum of the squared distances of each data point to each cluster center, if we use the Euclidean distance, weighted by the membership of the data point to each cluster, for all data points.

Assuming we want to subdivide the scene into c clusters, we compute a sum of dispersion between the data points x_k and a set of prototypes (cluster center points) $v_1, v_2, \ldots, v_c$:

$$Q = \sum_{i=1}^{c} \sum_{k=1}^{n} u_{ik}^{p} d(x_k, v_i) \tag{4.9}$$

with $d(x_k, v_i)$ being a given fixed distance function (e.g. Euclidean distance, or any l_p-Norm in general, see Table 4.1) between the data points x_k and v_i, the center point of cluster i.

Furthermore, Eq. (4.9) contains the fuzziness factor p, $p > 1$, and a partition matrix $U = [u_{ik}]$, $i = 1, 2, \ldots, c$, $k = 1, 2, \ldots, n$, which allocate the data points to the clusters. A fuzziness factor $p = 1$ means that the algorithm is doing a hard clustering, like the c-means algorithm, and if $p \to \infty$ the membership will be equal in all clusters. The fuzzy clustering algorithm will iteratively optimize Eq. (4.9). In each iteration, all elements u_{ik} of the partition matrix U are updated using Eq. (4.10).

$$u_{ik} = \frac{1}{\sum_{j=1}^{c} \left(\frac{d(x_i, v_k)}{d(x_i, v_j)} \right)^{\frac{2}{p-1}}} \tag{4.10}$$

In the next step the algorithm updates the cluster centers v_k:

$$v_k = \frac{\sum_{i=1}^{n} u_{ik}^{p} \cdot x_i}{\sum_{i=1}^{n} u_{ik}^{p}} \tag{4.11}$$

The algorithm repeats these steps until the movement of the center point of all clusters is smaller than a predefined stop criterion. The time complexity of fuzzy c-means clustering algorithm is $\mathcal{O}(n \cdot c^2 \cdot d \cdot$

r_{iter}), where n is the number of data points, c the number of clusters, d the number of dimensions and r_{iter} the number of iterations.

4.5.1 *Stopping Criterion*

The stopping criterion is used to determine if the clustering process has reached a steady state and further investigation will not improve the result anymore. Therefore, in most cases, the partition matrices of two successive iterations are compared. If the value of $\|U(run+1) - U(run)\|$ is smaller than a predefined threshold ε, the clustering process stops. To be more specific, the biggest change in the partition matrix can be used:

$$\|U(run+1) - U(run)\| = \max_{i,k} |u_{ik}(run+1) - u_{ik}(run)| \tag{4.12}$$

The value of the threshold ε depends on the underlying application; normally a value between 10^{-3} and 10^{-5} is used.

4.5.2 *Advantages and Disadvantages*

The fuzzy version of the c-means algorithm has the same advantages and disadvantages like the standard c-means approach (see Section 4.4.2), because with a fuzziness factor $p = 1$ the fuzzy version will perform the same clustering results. The fuzzy c-means algorithm has one particular advantage: one data point can belong to more than just one cluster, which is very important for our collision detection approach (see Section 5.1.2).

4.6 EXCURSUS: OUR NEW BATCH NEURAL GAS APPROACH FOR HIERARCHY CONSTRUCTION

Another clustering algorithm is Neural Gas (NG). The basic algorithm of the NG uses a special version of the c-means algorithm (Section 4.4), which takes neighborhood ranking into account [Cot+06; MBS93]. Cottrell, Hammer, Hasenfuß, and Villmann [Cot+06] introduced a batch version of the NG, so-called Batch Neural Gas (BNG), which shows much faster convergence than the standard NG approach.

BNG is a very robust clustering algorithm, which can be formulated as stochastic gradient descent with a cost function closely connected to quantization error. Like c-means, the cost function minimizes the mean squared Euclidean distance of each data point to its nearest center. But unlike c-means, BNG exhibits very robust behavior with respect to the initial cluster center positions (the prototypes): they can be chosen arbitrarily without affecting the conver-

gence. Moreover, BNG can be extended to allow the specification of the *importance* of each data point.

In the following we will give a quick recap of the basic BNG algorithm. Given points $x_j \in \mathbb{R}^d, j = 0, \ldots, n$ and prototypes $v_i \in \mathbb{R}^d, i = 0, \ldots, c$ initialized randomly, we set the rank for every prototype v_i with respect to every data point x_j as:

$$k_{ij} := |\{v_k : d(x_j, v_k) < d(x_j, v_i)\}| \in \{0, \ldots, c\} \tag{4.13}$$

In other words, we sort the prototypes with respect to every data point. After the computation of the ranks, we compute the new position for each prototype:

$$v_i := \frac{\sum_{j=0}^{n} h_\lambda(k_{ij}) x_j}{\sum_{j=0}^{n} h_\lambda(k_{ij})} \tag{4.14}$$

These two steps are repeated until a predefined stop criterion is met. In the original publication by Cottrell, Hammer, Hasenfuß, and Villmann [Cot+06] a fixed number of iterations is proposed. Indeed, after a certain number of iteration steps, which depends on the number of data points, there is no further improvement.

The convergence rate is controlled by a monotonically decreasing function $h_\lambda(k) > 0$ that decreases with the number of iterations t. We decide to use the function proposed in the original publication [Cot+06]: $h_\lambda(k) = e^{-\frac{k}{\lambda}}$ with initial value $\lambda_0 = \frac{c}{2}$, and reduction $\lambda(t) = \lambda_0 \left(\frac{0.01}{\lambda_0}\right)^{\frac{t}{t_{max}}}$, where t_{max} is the maximum number of iterations. These values have been taken by Martinetz, Berkovich, and Schulten [MBS93] also.

4.6.1 *Batch Neural Gas for BVH Construction*

Like we mentioned before, in computer graphics objects are usually represented only by their *surface*, e. g., by a polygonal mesh or as an implicit NURBS surface. Consequently, most work on BVHs have been spent on these object representations. Recently, Weller and Zachmann [WZ09] presented a new *volumetric* method to represent 3 D object by a sphere packing, called Inner Sphere Trees (ISTs). The basic idea is to fill a typical 3 D surface representation from the *inside* with a set of *non-overlapping* spheres with different radii. These inside sphere packings allow the computation of the *penetration volume* as penetration measure for collision queries, if two objects are intersect. According to Fisher and Lin [FL01, Section 5.1], this penetration measure is "the most complicated yet accurate method" to define the extent of intersection.

However, constructing a BVH of such sphere packings is a difficult task, because traditional methods that are optimized for surface representations are not automatically also suited for a volumetric BVH.

For instance, BVH construction methods that were designed for classical *outer* sphere trees, like the medial axis approach [BO04; Hub95] work well if the spheres constitute a *covering* of the object and have very similar size, but in the scenario of ISTs they use disjoint inner spheres that exhibit a large variation in size. Other approaches based on the *k-center problem* work only for sets of points and can be hardly extended to spheres.

For the construction of BVHs on sphere packings, we extend the classical BNG clustering method. In its pure form, BNG partitions a set of data points into a pre-defined number of clusters by minimizing the mean squared Euclidean distance of each data point to its nearest center (prototype). Furthermore, we adopt an extension called *magnification control* that enables us to take also the spheres' volume into account.

The BNG algorithm adds a single point, a so-called *prototype*, for each cluster to the sphere packing and moves them iteratively until some convergence criterion is met. The movement depends on the distance of the prototypes to all data points. Unfortunately, this procedure requires a lot of convergence steps and consequently, is relatively slow. Moreover, we have to start such a time consuming step for each BV in the BVH individually. Even if the construction of the BVH is a pre-processing step that has to be done only once, it should not waste too much time. For instance, if we want to add one or more new objects to an interactive real-time simulation we usually do not want to wait for minutes until the BVH for these objects have been constructed.

To eliminate this limitation and to construct a BVH in a few seconds, we developed a novel massively parallel version of BNG that is especially optimized for the construction of BVHs. Our novel algorithm for BVH construction runs entirely on the GPU and in general, it does not require any time consuming communication between CPU and GPU during the whole BVH construction. Moreover, it reduces the theoretic complexity of the hierarchy construction of $\mathcal{O}(n \cdot \log n)$ for the CPU version to $\mathcal{O}(\log^2 n)$ using only a linear number of processors $\mathcal{O}(n)$. Our novel parallel approach is easy to implement and robust against the start positions of the prototypes, therefore it will not stuck in local optima. Our CUDA implementation shows a significant speed up compared to the CPU version (see Section 4.6.3). Moreover, our results show that the BNG-based BVHs perform much better than BVHs that are constructed using simple heuristics for the sphere partitioning.

Magnification Control

In its pure version, the BNG only takes the location of the centers of the spheres into account. Experiences have shown that this already produces reasonable results for the collision detection query perfor-

mance. However, this procedure does not yet take the radii of the spheres into account. This is, because BNG uses only the *number* of data points and not their *importance* for the hierarchy construction process. The result of this is that the prototypes tend to avoid regions that are covered with a very large sphere, due to the fact that, centers of big spheres are recognized as outliers and therefore, they are placed on very deep levels in the hierarchy. It is far better, however, to place bigger spheres at higher levels of the hierarchy in order to get early lower bounds during distance traversal.

Therefore, we have to use an extended version of the classical BNG, which takes the size of the spheres into account, to produce a more efficient BVH. Our extension is based on an idea of Hammer, Hasenfuss, and Villmann [HHV06], where *magnification control* is introduced. The main idea is to add weighting factors for each data point in order to "artificially" increase the density of the space in certain areas. With weighting factors $vol(x_j)$, Eq. (4.14) becomes

$$v_i := \frac{\sum_{j=0}^{n} h_\lambda(k_{ij}) vol(x_j) x_j}{\sum_{j=0}^{n} h_\lambda(k_{ij}) vol(x_j)} \tag{4.15}$$

where $vol(x_j)$ represents a control parameter to take care of the importance of a given data point x_j. Hammer, Hasenfuss, and Villmann [HHV06] use a function of density to control the magnification. Due to the fact that, IST use a sphere packing where all spheres are disjoint, we already know the density. Therefore, we can directly use the volumes of each sphere as weighting factor: $vol(x_j) = \frac{4}{3}\pi r^3$.

Batch Neural Gas-based Hierarchy Creation

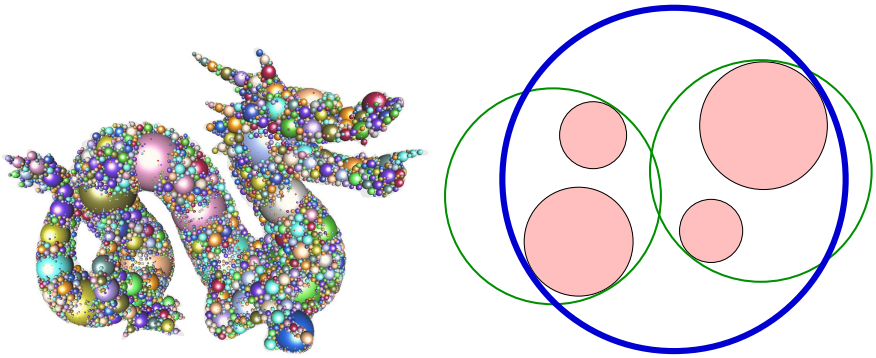


Figure 4.3: On the left: Dragon model with the corresponding dense polydisperse sphere packing. On the right: Some spheres are wrapped in a hierarchy, the parent sphere (blue) covers all its leaf nodes (red), but not its direct children (green).

As we noted in Section 4.6.1, the IST use a volumetric polydisperse sphere packing representation of the 3 D objects. In a polydisperse sphere packing all spheres are located completely *inside* the object,

and they *do not overlap* each other. However, the radii of the spheres can vary from sphere to sphere. This characteristic allows us to approximate the object's volume to any required accuracy—defined by the smallest radii—by using *space-filling* sphere packings.

This sphere packing is used to create an *inner* BVH where the inner spheres are the leaves of the hierarchy. In order to construct our hierarchy we use a top-down *wrapped hierarchy* approach. According to the notion of Agarwal, Guibas, Nguyen, Russel, and Zhang [Aga+04], in a top-down *wrapped hierarchy* inner nodes are tight BVs for all their leaves, but they do not have to bound their direct children (see Figure 4.3). In contrast to layered hierarchies, a wrapped hierarchy produces tighter BVs for their inner nodes. In a top-down hierarchy construction procedure, you begin at the root node of the hierarchy, which covers all inner spheres and divide these part into several subsets.

4.6.2 *Batch Neural Gas Hierarchy Construction*

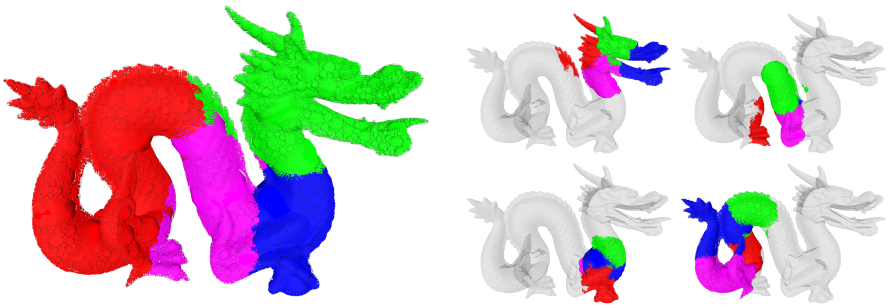


Figure 4.4: This Figure depicts a BVH constructed with our new BNG clustering algorithm with magnification control. The left image shows the partitioning of the root sphere. Each of the 4 clusters of the root node are shown in the right images with its corresponding clustering. All inner spheres that are colored with the same color are assigned to the same bounding sphere.

Summing up the just explained findings, one can describe the BNG-based BVH creation process as follows: in the first step we compute one bounding sphere for all inner spheres (inner spheres from sphere packing, which are the leaves in the hierarchy), and store this sphere as root node of the hierarchy. Therefore, we use an algorithm proposed by Gärtner [Gär99] to compute the smallest enclosing sphere. The next step is to subdivide the set of inner spheres into subsets. For this procedure we use an extended version of the BNG algorithm with magnification control. This procedure is repeated until a predefined criterion is met, i. e., number of inner spheres the bounding sphere encapsulate (Figure 4.4 depicts an example of hierarchical ISTs).

Parallel Hierarchical Batch Neural Gas

Like Figure 4.4 depicts, our extend version of the BNG algorithm produces a very good partitioning of the inner spheres, however the serial implementation is very slow. The current version of the BVH algorithm executes $\mathcal{O}(n)$ BNG calls—a call has to be executed for each hierarchy sphere—where n represents the number of inner spheres and we want one sphere per leaf. Suppose we have a balanced tree with a depth of $\mathcal{O}(\log n)$, which results in an total running-time of $\mathcal{O}(n \cdot \log n)$. It should be noted that there is a very high hidden constant value, which results from the number of the BNG iteration steps.

A closer look at the BNG algorithm in its pure form, but also the hierarchical BNG calls of our BVH creation method, depicts that these steps are perfectly tailored for parallelization [Wel+14]. Using a linear number of processors $\mathcal{O}(n)$, it is possible to reduce the asymptotic running-time to $\mathcal{O}(\log^2 n)$. In the following section we give an overview of our parallel hierarchical BNG implementation running entirely on the GPU.

The rank for every prototype and the subtotal sum $h_\lambda(k_{ij})vol(x_j)x_j$ and $h_\lambda(k_{ij})vol(x_j)$ can be computed completely independently for each sphere x_j, on the first level of our hierarchy. By using a parallel scan algorithm [BH11; SHG08] we can compute the total sums $\sum_{j=0}^{n} h_\lambda(k_{ij})vol(x_j)x_j$ and $\sum_{j=0}^{n} h_\lambda(k_{ij})vol(x_j)$ in parallel too. The decision about which spheres are assigned to which prototype can be done in parallel as well. Therefore, we have to compute all distances between a sphere and all prototypes and choosing the nearest prototype for the selected sphere. We perform a hard clustering, so each sphere is assigned to exactly one prototype.

To create the next level of the BVH we have to insert new prototypes for each prototype of the previous hierarchy level. The number of new prototypes is given by the branching factor of the tree, i. e., a branching factor of 4. Starting an own CUDA thread for each subset of spheres will destroy the advantages of parallel computing. We will discuss this approach in detail below.

The first step is to sort the spheres with respect to the prototype that each sphere was assigned to (see Figure 4.5). Therefore, we use a parallel sorting algorithm [SHG09], e. g., from the Thrust library [BH11]. This procedure achieves that we can use fast parallel prefix sum computations in the later steps. Once the sorting step we insert 4 new prototypes for each prototype from the previous hierarchy level. Because each sphere is exactly assigned to one prototype in the previous level allows us to compute all the values that are required by BNG, e. g., the rank, the subtotal sums $h_\lambda(k_{ij})vol(x_j)$ and $h_\lambda(k_{ij})vol(x_j)x_j$, in parallel per sphere. It is necessary to ensure that these values are computed for the correct prototypes, the new added ones (see Figure 4.6).

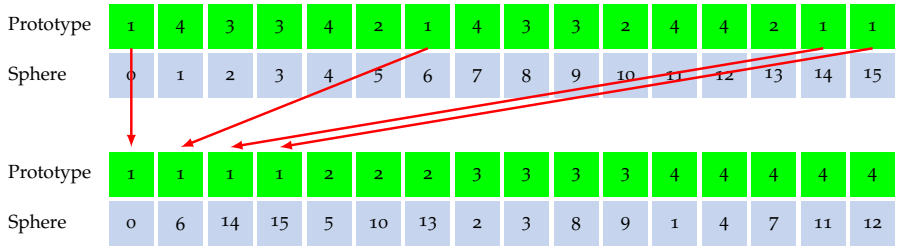


Figure 4.5: The top two arrays show the indices of the spheres and the prototype each sphere is assigned to after the initialization of the BNG clustering. The next step is to sort the array of prototypes, while we move the assigned spheres in the same way. Note, that each sphere is assigned to exactly one prototype.

The last step is to sum up all subtotal sums to compute the new position of the prototypes. As you will see from Eq. (4.15) we have to compute $\sum_{j=0}^{n} h_\lambda(k_{ij})vol(x_j)x_j$ and $\sum_{j=0}^{n} h_\lambda(k_{ij})vol(x_j)$. Fortunately, we can use the results of the parallel prefix sum [BH11; SHG08] we performed before. To compute the sums for each new prototype, we have to subtract the prefix sum results at the borders of our sorted prototype array, like we demonstrate in Figure 4.7.

Algorithm 4.4 provides a short overview of our complete parallel hierarchical BNG implementation.

Algorithm 4.4 Parallel hierarchical BNG

```
while not on inner sphere level do
    iteration = 0
    while iteration < maxNumberIterations do
        iteration++
        in parallel sort prototype array
        for all Spheres do in parallel
            compute h_λ(k_ij)vol(x_j)x_j and h_λ(k_ij)vol(x_j)
        end for
        in parallel compute prefix sum
        for all Prototypes in level do in parallel
            compute new position
        end for
    end while
end while
```

The parallel prefix sum and the parallel sorting of the prototypes for n inner spheres can be computed with a linear number of processors $\mathcal{O}(n)$ in $\mathcal{O}(\log n)$ time. However, both algorithms are based on an implicit balanced binary tree structure (for more details we refer to [SHG09] and [SHG08]). All the "per sphere" operations of the Algorithm 4.4 have a complexity of $\mathcal{O}(1)$. In the case of a balanced tree, the outer while-loop has to be called $\mathcal{O}(\log n)$ times. This leads to an overall parallel running-time complexity of our algorithm of $\mathcal{O}(\log^2 n)$. The memory consumption is $\mathcal{O}(n)$ because we have a predefined number of spheres, depending on the density of the sphere packing.

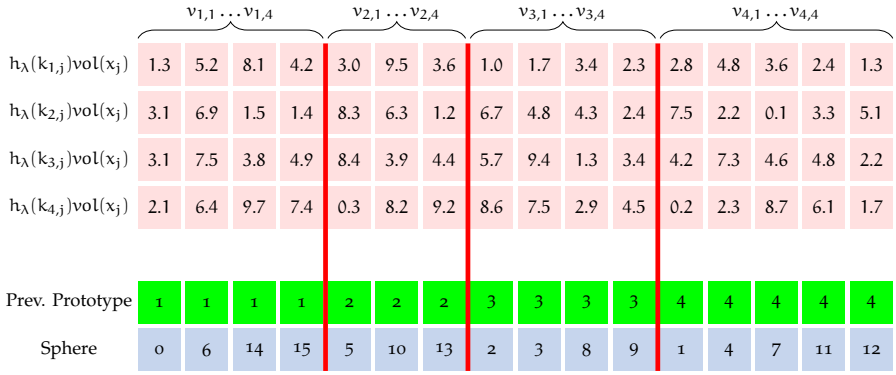


Figure 4.6: This Figure shows an example of a deeper level of the hierarchical BNG. Like we show in Figure 4.5, each sphere is assigned to exactly one prototype. For each prototype $v_1, \cdots, v_4$ from the previous level, we add 16 new prototypes, $v_{1,1}, \cdots, v_{4,4}$ and compute all the values that are required by BNG, e. g., the rank, the subtotal sums $h_\lambda(k_{ij})vol(x_j)$ and $h_\lambda(k_{ij})vol(x_j)x_j$. Because of the fact that each sphere is exactly assigned to one prototype we can re-use the memory allocated for the previous hierarchy level. Thus, we have a constant number of memory usages because for each level of the hierarchy we have a predefined number of prototypes. Consequently, no memory transfer between CPU and GPU is necessary.

4.6.3 *Results*

We implemented our algorithms using C++ for the CPU version and CUDA for the GPU version. All tests were performed on an Intel I7 CPU with 8 GB main memory and a NVIDIA Gefore GTX 780 GPU with 3 GB of memory.

We used complex 3 D models with very different shapes in our timings: in particular, two animal models, a detailed model of the human brain and a statue (see Figure 4.8). Additionally, we filled all models with different numbers of spheres ranging from 2 k up to 100 k inner spheres.

Our results show, that our novel hierarchical BNG hierarchy creation algorithms outperforms the CPU significantly. More precisely, we get an acceleration of a factor of 15 for all objects (see Figure 4.9). Please note, that our algorithm is not optimized yet, i.e. we do not use advanced CUDA acceleration techniques like shared memory. In practice it is essential that there is not too much traffic between the memories of the CPU and the GPU. In our algorithm there is almost no traffic required. In our current implementation, we only have to save the positions of the prototypes from the last iteration in the outer loop of Algorithm 4.4. However, this is also not really necessary. In the future, we plan to move the smallest enclosing sphere computation to the GPU too. Then, we only have to read back the whole

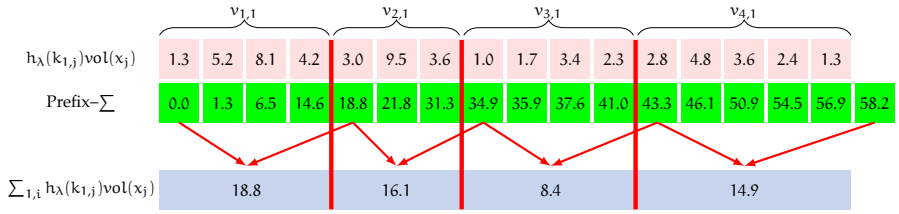


Figure 4.7: In order to update the position of the prototypes for the next iteration in the BNG algorithm, we have to compute $\sum h_\lambda(k_{ij})vol(x_j)$ and $\sum h_\lambda(k_{ij})vol(x_j)x_j$. To realize this we compute the prefix sum (green array) for each of the four prototype arrays from Figure 4.6. By subtracting the values at the borders will directly deliver us the individual sum for each prototype.

Figure 4.8: The objects we used in our timings: a cow, a pig, a human brain and a statue.

hierarchy once. We only have to allocate memory for the prototypes once. This memory can be re-used for all iterations.

We also tested the performance of our BNG-based hierarchies for collision detection queries. To do that, we implemented two simple competing partitioning heutistics: First, we greedily choose the four biggest spheres and partition the smaller spheres to the closest of these large elements. Second, we sorted the spheres with respect to the coordinate axis and choose the two axes with the largest extend. Again, we assigned the intermediate spheres to the closest of the four extreme spheres.

In our two test scenes (see Figure 4.10) we used penetration volume queries. All tests were run on an Intel I7 processor. The collision query algorithm uses hand optimized SIMD code. The results show that the BNG hierarchies performed best in our entire query test runs. Actually, they are more than a factor of 4 faster than the greedy choice of outer spheres (see Figure 4.11). Surprisingly, the simple greedy choice of biggest spheres performs well, but it is still 20 % slower than our BNG hierarchies.

4.6.4 *Improvements of Batch Neural Gas for Hierarchy Construction*

Our novel approach also opens up several avenues for future work. In the previous section we already mentioned the planned parallel implementation of the minimum enclosing sphere computation. How-

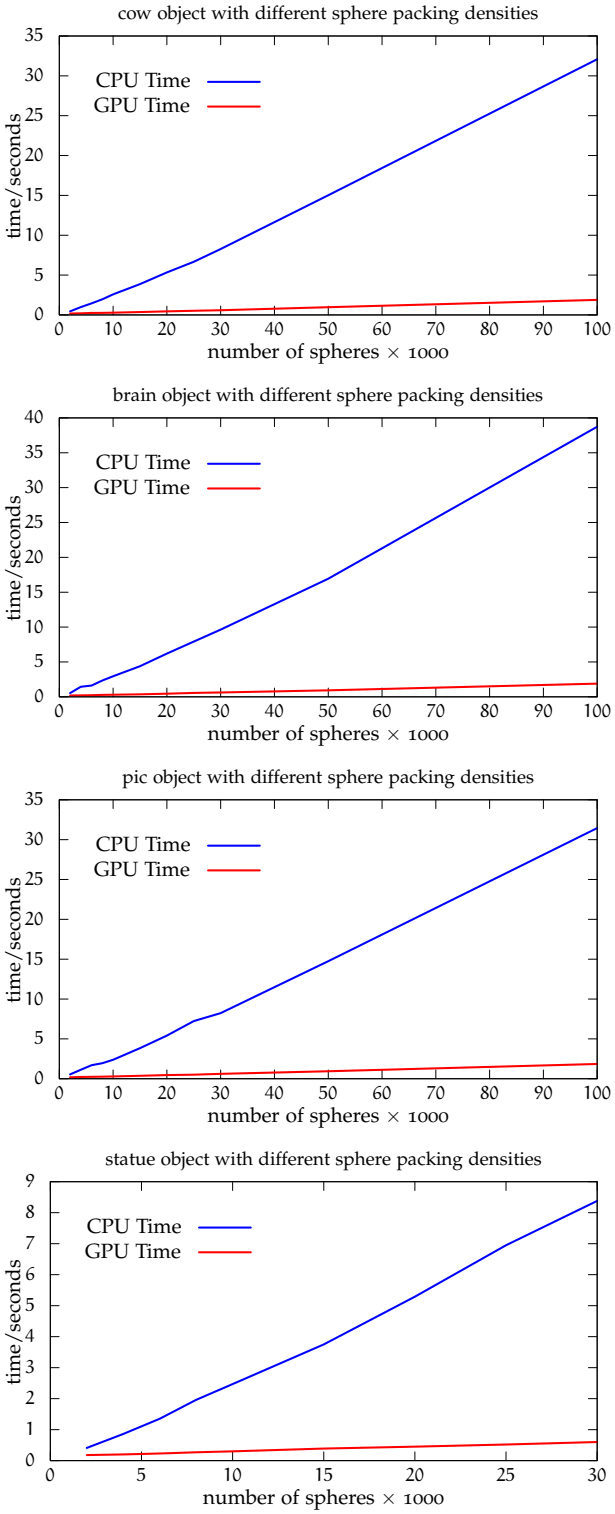


Figure 4.9: CPU and GPU time for the BVH construction for some models (see Figure 4.8) with different sphere packing densities.

ever, it would be also interesting to apply our algorithm to other volumetric object representations than sphere packings, e. g., tetrahedra

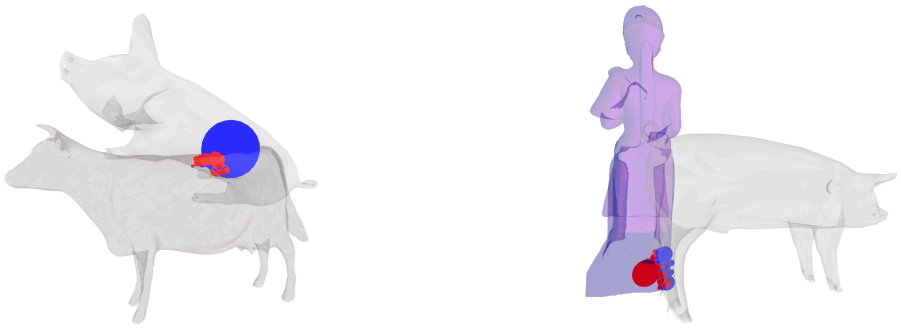


Figure 4.10: The test scenes for collision detection queries. On the left: cow and pig. On the right: pig and statue.

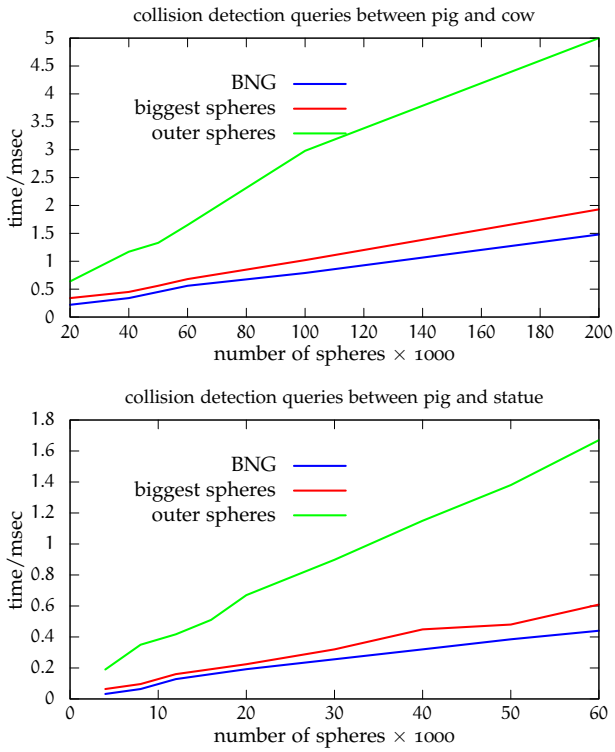


Figure 4.11: On the left: Average time for collision detection queries in the pig and cow scene. On the right: the same for the pig and statue scene (see Figure 4.10).

or ellipses. This could improve the quality of the volume covering because spheres do not fit well into some objects, especially if they have many sharp corners or thin ridges. Another option could be the investigation of our clustering-based BVH construction for classical *outer* BVHs. Currently, most implementations of classic BVHs use traditionally a branching factor of two. Due to recent developments in CPU technologies like SSE and Advanced Vector Extensions (AVX), higher branching factors could accelerate queries significantly. However, in this case, also more sophisticated partitioning techniques for the BVH construction are required because traditional heuristics for

binary trees may not work anymore. Finally, we would like to explore other uses of *inner bounding volume hierarchies*, such as ray tracing or occlusion culling. Note that the type of Bounding Volume chosen for the "inner hierarchy" probably depends on its use.

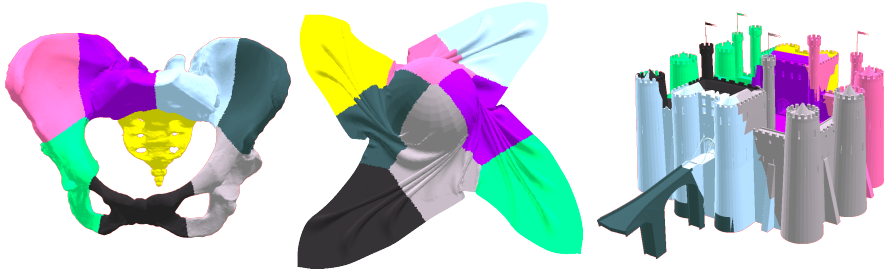
4.7 FUTURE WORK

The creation of a good partitioning is very important to balance the load onto different GPUs. Therefore, a good cluster generation is unavoidable. Like we mentioned before the quality of the c-means clustering algorithm depends on the cluster initialization, because this approach can run in a local optima. The BNG algorithm instead generates high quality cluster independently from the initialization process. The downside of this approach is the high runtime.

In our case, a combination of both approaches can improve our collision detection approach further. Therefore, we can use the BNG algorithm for the initialization step to generate a high quality subdivision of the scene. Because this step will be performed at the beginning of the simulation only, the higher runtime can be neglected. In the time critical phase, between the simulation steps, our data points (primitives within the scene) are moving over the time. For the update process of the cluster membership we can now use the very fast fuzzy c-means clustering approach. A combination of both approaches will generate a high quality partitioning and will be runtime sensitive.

5

OUR NOVELL COLLISION DETECTION APPROACH BASED ON FUZZY SCENE SUBDIVISION



In the chapters before, we have introduced some methods like scene subdivision using fuzzy clustering or the SaP algorithm to speedup the collision detection process. In this chapter, we will use these techniques to define a novel collision detection approach for rigid and deformable objects.

Our collision detection algorithm is completely executed on the GPU and especially well-tailored to use the massively parallel performance of the device. With our method we can handle the broad phase as well as the narrow-phase within one single framework. Our collision detection algorithm works directly on all primitives of the whole scene, which results in a simpler implementation and can be integrated much more easily by other applications. Due to the fact that we are working on the primitives directly, no approximation errors occur. Furthermore, we can compute inter-object and intra-object collisions of rigid and deformable objects consisting of many tens of thousands of triangles in a few milliseconds on a modern computer. We have verified the performance of our novel collision detection with commonly used benchmarks (see Section 5.6). Furthermore, we have integrated our approach into Bullet, a widely used physics engine (see Section 6.5). Additionally, we present our novel Benchmarking Suite for rigid body collision detection in Section 5.7. With this Benchmarking Suite it is possible to compare the time needed for a collision detection and the quality of the computed force and torque.

5.1 SCENE SUBDIVISION

For the subdivision process of the virtual 3 D scene into independent parts, we use a clustering algorithm, to be more precise, fuzzy c-means. In Section 4.5 we gave an insight into the functionality of this algorithm. The first question that needs asking here is why we are using a soft clustering algorithm, although a soft clustering has a

higher complexity than a hard clustering algorithm. We use a fuzzy clustering algorithm because the primitives, who are located on the border between two neighboring clusters, have to be in both clusters. If adjoining clusters are not connected, then in some cases collisions across the border of the clusters would not be taken into account (see Figure 5.13 and Section 5.1.2 for further information).

Another advantage—why we choose this algorithm for the clustering step—is that the fuzzy c-means algorithm can be run incrementally thus exploiting temporal coherence that is inherent in most realistic scenes. For the next iteration the algorithm uses the last computation result as starting point and iteratively minimizes the total error with the new data points. This approach takes advantage of the fact that the scene changes not very much from one frame to the next one. Therefore, this algorithm is especially tailored for simulations where objects are moving on a path from one place to another within the scene, like in the most real-world scenarios.

Algorithm 5.5 Scene Subdivision

```
Input: list L of primitives of all objects, number of cluster c
Output: clustering
  function computeClustering(c)
    for all primitives in L do in parallel
      compute center of mass        ▷ for every primitive ONE data point
    end for
    for all data points do in parallel
      do clustering & generate partitioning matrix (membership values)
    end for
    for all elements in partitioning matrix do in parallel
      map primitive to cluster(s)                  ▷ one or more clusters
    end for
  end function
```

Algorithm 5.5 provides a short overview of the scene subdivision process. In the first step we generate data points to represent the primitives. In following steps these data points are used to classify the primitives into a given number of clusters. In the last step the membership value is used to determine to which clusters a primitive belongs to. It should be noted that a primitive can be assigned to more than one cluster.

The following sections describe each stage of the Algorithm 5.5 in greater detail.

5.1.1 *Data Points for the Scene Subdivision Process*

A clustering process groups data points into a predefined number of subsets, called clusters. Due to the fact, that we use triangles as primitives, we decide to use the center of mass of each triangle to represent the primitive, which becomes our data points. With this step the data points for clustering was threefold decreased. For other types of primitive—than a triangle—it can be possible that the cen-

ter of mass does not represent the primitive very well. In that case another representation is needed. The computation of the center of mass can be performed in parallel perfectly. Furthermore, we have a full coalesce memory access pattern (see Figure 3.3 for different types of memory access pattern), while reading the coordinates for each primitive, which results in a very fast computation process.

5.1.2 *Clustering Process*

(a) Armadillo is subdivided into 2 clusters.

(b) Human brain is subdivided into 10 clusters.

Figure 5.2: Example of object subdivision using fuzzy c-means clustering algorithm.

Like we mentioned before we use a soft clustering algorithm, fuzzy c-means (see Section 4.5 for more details), for the subdivision process. We use all center of mass points as data points and subdivide the scene into a given number of clusters. Figure 5.2 shows some examples of object subdivision process with a soft clustering algorithm. Due to the fact that we use a fuzzy clustering algorithm some primitives can be assigned to more than one cluster. Primitives on the border between two clusters have a smaller membership value (ranged between 0 and 1), which means that they do not belong to one cluster only. We assign a primitive to a cluster, if the membership value is higher than a precomputed threshold.

Why soft clustering?

In Chapter 4 we introduced some clustering methods, a soft and a hard clustering approach. The question also arises of whether we are really have to use a soft clustering approach, or is it sufficient if we use a hard clustering, which performs faster. For example, the complexity of c-means clustering is in $\mathcal{O}(n \cdot c \cdot d \cdot r_{iter})$, while fuzzy c-means has a complexity in $\mathcal{O}(n \cdot c^2 \cdot d \cdot r_{iter})$, where n is the number of data points, c the number of clusters, d the number of dimensions and r_{iter} the number of iterations.

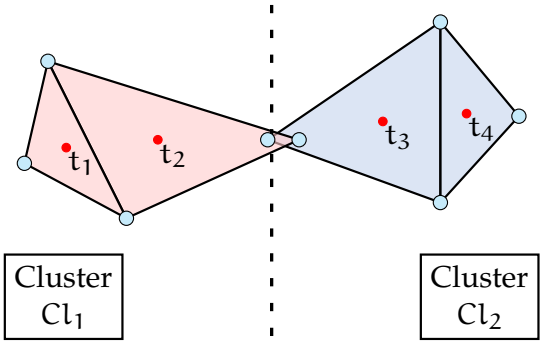


Figure 5.3: A simple scene—consisting of 4 primitives (triangles)—is subdivided into 2 clusters, Cl_1 and Cl_2, by a hard clustering algorithm. Primitives assigned to Cl_1 are colored in light red and primitives assigned to Cl_2 are colored in blue. A red marker depicts the center of mass. All center of mass points are used as data points for the clustering process.

Suppose we use a hard clustering approach, like c-means, and we use the center of mass of our primitives as data points. Now we subdivide a scene into 2 clusters, for example see Figure 5.3. In this example two primitives, triangle t_1 and t_2, are assigned to cluster Cl_1 (primitives in light red) and two primitives, triangle t_3 and t_4, are assigned to cluster Cl_2 (primitives in blue). Our collision detection approach will now perform all tests for all clusters independently. This means that we have to perform one intersection test for primitives t_1 and t_2, and one intersection test for primitives t_3 and t_4. Primitives assigned to different clusters are *not* tested for intersection and therefore, the intersection between primitive t_2 and t_3 will be missed. To prevent this error from occurring, you have to check primitives on the border between adjacent clusters for intersection too.

This is precisely the reason why we use a soft clustering for our scene subdivision process, because we want an exact collision detection approach, which will not miss any collisions, and therefore, we have to take intersections of primitives across the cluster border into account.

Choosing the Number of Clusters

Determining a good number of clusters depends on the underlying data set or, in our case, the 3 D scene. If the number of clusters is to low then big parts of a scene are checked for collisions and this can lead to unwanted false-positives in the PCA step of our collision detection pipeline (see Section 5.2.1). On the other hand, a higher number of clusters increases the number of primitives, which are assigned to more than one cluster and that will leads to a higher collision detection processing time, because the computation time of the clustering process will increase. The overall complexity of the clustering algorithm is in $\mathcal{O}(n \cdot c^2 \cdot d \cdot r_{iter})$, where n is the number of data

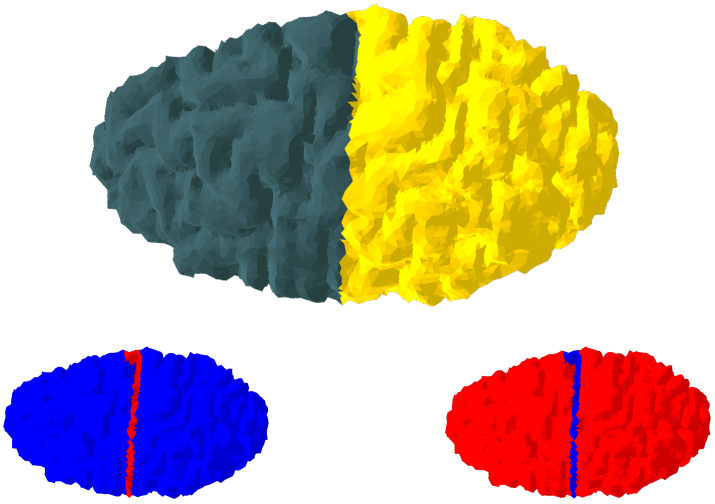


Figure 5.4: Human brain is subdivided into 2 clusters (figure on the top). Primitives which belong to *exactly* 1 (bottom left) or 2 clusters (bottom right) are colored in blue.

points, c the number of clusters, d the number of dimensions and r_{iter} the number of iterations.

Figure 5.4 shows a 3 D model of a human brain. In this example we set the number of clusters to 2. Figure 5.4 depicts that only primitives on the border of the two adjoining clusters are assigned to both clusters. So the overall number of primitives in all clusters only increases by a minimum number.

In contrast to Figure 5.4 we now use 4 clusters to subdivide the human brain, see Figure 5.5. This Figure shows that there are only a few primitives assigned to exactly one cluster, much more primitives are assigned to 2 or 3 clusters. So the overall number of primitives, which are assigned to the clusters, increase strongly compared to the example where we only use 2 clusters.

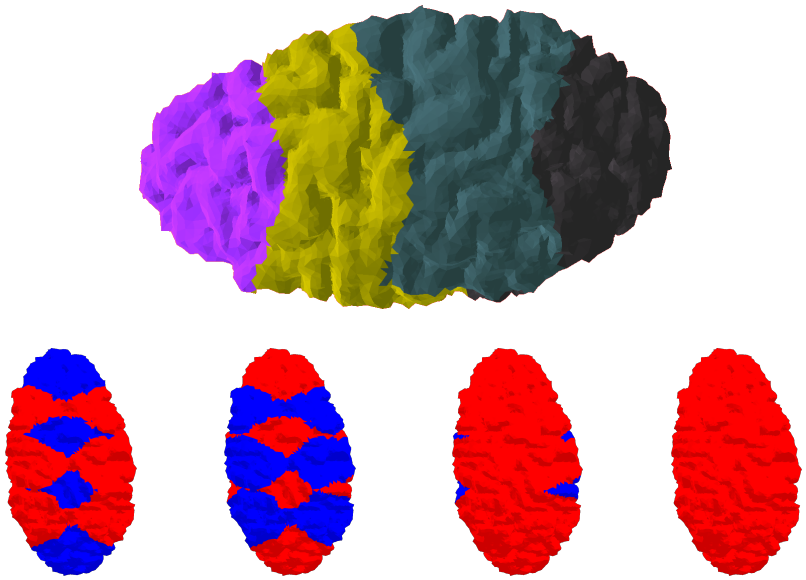


Figure 5.5: Human brain is subdivided into 4 clusters (figure on the top). Primitives which belong to *exactly* 1, 2, 3 or 4 clusters (from right to left) are colored in blue.

Like we mentioned before, the number of clusters directly influences the performance of our collision detection approach. Figure 5.6 shows the collision detection time needed for some models with different configurations for the number of clusters.

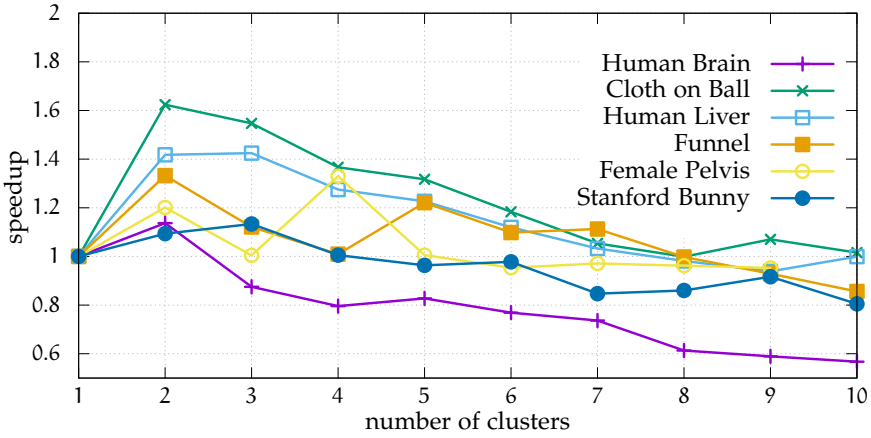


Figure 5.6: Influence of the number of clusters on the collision detection time needed for different models, e. g., human brain, Cloth on Ball and Funnel benchmarking models, human liver, female pelvis and the Stanford Bunny.

Our collision detection algorithm performs best if we choose 2 clusters for the human brain model. It should be noted that this scene do not change and therefore, there is no need to change the number of clusters over the time.

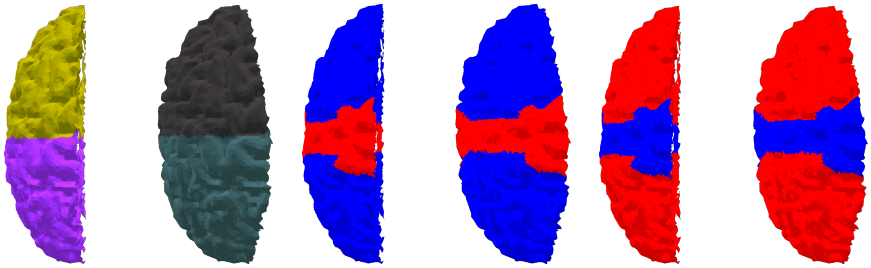
Let us assume that a scene consisting of 2 objects with a certain distance between them. Then the scene should be subdivided into 2 clusters at least (see Figure 5.7).

(a) 2 human brain models are subdivided into 2 clusters.

(b) Primitives which belong to exactly one cluster are colored in blue.

Figure 5.7: A scene consisting of 2 human brain models, which is subdivided into 2 clusters. The illustration on the left shows that all primitives from a model are assigned exactly to one cluster.

Using 4 clusters improves the speed of the collision detection algorithm again. As you can see in Figure 5.8 each object is subdivided into two parts. Thus we eliminate the limitations of the PCA process (see Section 5.2.1). Furthermore, only a few numbers of primitives on the border between the clusters are added into more than one cluster.



(a) 2 human brain models are subdivided into 4 clusters.

(b) Primitives which belong to exactly one cluster are colored in blue.

(c) Primitives which belong to exactly 2 clusters are colored in blue.

Figure 5.8: A scene consisting of 2 human brain models, which is subdivided into 4 clusters. The illustrations show that each brain is subdivided into exactly 2 parts. Thus, a primitive can belong to a maximum of 2 clusters.

Mapping Primitives to Clusters

The partitioning matrix contains all membership values indicating the assignment of a primitive to each cluster. If the scene is subdivided into 2 clusters and a primitive has a membership value of 0.5 for both clusters, this means that this primitive lies precisely at the middle of both clusters. Therefore, this primitive has to be added to both

clusters because it is located on the border between these clusters (see Figure 5.13). To determine if a primitive is located on the border or not the number of clusters is important. Like you can see the membership value and the assignment to the clusters depends on the number of clusters the scene is subdivided in. We use a simple formula to determine if a primitive is added to a cluster or not:

$$\text{threshold} = \frac{1}{c} - \frac{1/c}{10} \tag{5.1}$$

where c is the number of clusters. If the membership value for a cluster is bigger than this threshold we assign the primitive to this cluster.

Future Work

In the current implementation the number of clusters will not change. However, this is not perfect in any cases. If the simulation change, i. e., if a scene consists two objects and both objects are moving away from each other, then both objects should be in different clusters and in most cases each object should be subdivided again, so that we have 4 clusters at least. The number of false-positives should also be considered as the indicator in order to decide whether the number of clusters should be decrease, increase or stay constant. This criterion is very important because with the subdivision technique we try to reduce the number of false-positives and therefore, this is a very good measure.

Furthermore, the length of borders between all clusters should be as short as possible. Thus, the number of primitives located on the border will decrease and therefore, the number of primitives, which are assigned to more than one cluster. The shape of all clusters and therefore, the length of the cluster borders can be controlled by the distance function (see Figure 4.2).

Like we already mentioned in Sections 4.4.2 and 4.7 the clustering results of the fuzzy c-means algorithm depends on the cluster initialization and perform poorly for non-global clusters, it gets stuck in a local optima sometimes. Even if we never get stuck in a local optima, we want to show up how you can compensate this disadvantage. To realize this you can perform the first clustering step with a clustering algorithm, which will not run in local optima, like BNG. For the first frame the higher running time can be neglected. After the initialization with BNG, which results in a high quality subdivision of the scene, we can use the much faster fuzzy c-means algorithm. This guarantees that our approach will be runtime sensitive and use an optimal clustering as starting point.

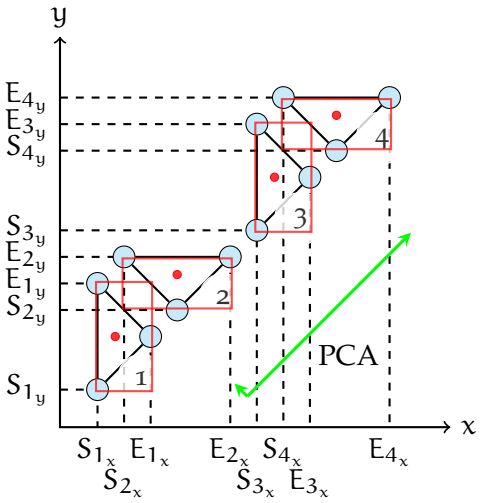
The process to determine whether a primitive is assigned to a cluster, depends on the fact if the membership value is greater than the threshold value or not (see Eq. 5.1). Currently, we use a very simple

equation, which gives opportunities for improvements. It would be preferable if the overlap between clusters is as small as possible and as large as necessary.

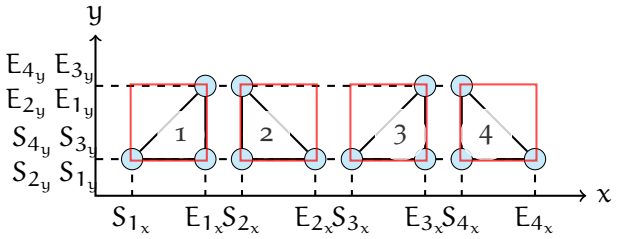
5.2 SWEEP-PLANE TECHNIQUE USING PCA

As we stated in Section 2.7 the SaP approach based on the relative position of primitives to each other. Therefore, this technique projects the boundary of all BVs onto all axes. If, for example, primitives are moving then in a significant amount of cases a huge number of false-positives may occur, when we choose any of the *fixed* world coordinate axes as sweep direction.

5.2.1 *Principal Component Analysis to Determine a Good Sweep Direction*



(a) The initial scene consisting of a number of triangles with corresponding BVs and the result of the PCA. As can clearly be seen the BVs of triangle 1 and 2, and triangle 3 and 4 intersect.



(b) Initial scene from Figure 5.9a, rotated so that the direction of the first component of the PCA points along the x-axis. As can clearly be seen, in this example the number of overlapping BVs reduced to zero.

Figure 5.9: Improvement of Sweep-and-Prune approach via Principal Component Analysis.

Figure 5.9a depicts an example of a downside of using BVs, like AABBs or OBBs. In our case, the best sweep direction is the one that allows projection to separate the primitives as much as possible. In order to achieve the best sweep direction, even if the primitives move through 3 D spaces, we compute the PCA [Jol05; Liu+10] in every frame, because the direction of the first principal component maximizes the variance of primitives, after projection [Wu92].

The type of covariance analysis we perform is commonly used for dimension reduction and statistical analysis of data [Eri05]. The covariance matrix $\mathbf{Cov} = [h_{ij}]$ for all data points $x_1, x_2, \ldots, x_n$ is given by:

$$h_{ij} = \frac{1}{n} \sum_{k=1}^{n} (x_{k,i} - mean_i) \cdot (x_{k,j} - mean_j), \tag{5.2}$$

with $mean_i$ and $mean_j$ is the mean of the i-th and the j-th coordinate value of all the data points.

In the next step we compute the eigenvalues λ_0, λ_1, λ_2, and the corresponding eigenvectors **u**, **v**, and **w** of the covariance matrix **Cov**. Computing the eigenvalues and eigenvectors of a matrix is in most cases a nontrivial task. Golub and Van Loan [GV12] presented a numerical technique, which is commonly used. For the decomposition step, to compute the eigenvalues and eigenvectors of the matrix, we use codes provided by Numerical Recipes in C++ [Pre+07]. With those values it is possible to determine the axis along which the data points have the largest variance. The three orthogonal eigenvectors **u**, **v** and **w** define a new coordinate system where the x-axis points along the axis with the largest variance. We can now transform the old coordinates $(x_{k,x}, x_{k,y}, x_{k,z})$ into the new coordinates $(x'_{k,x}, x'_{k,y}, x'_{k,z})$ by the linear transformation:

$$\begin{pmatrix} x'_{k,x} \\ x'_{k,y} \\ x'_{k,z} \end{pmatrix} = \begin{pmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{pmatrix} \cdot \begin{pmatrix} x_{k,x} \\ x_{k,y} \\ x_{k,z} \end{pmatrix} \tag{5.3}$$

After the transformation by the eigenvectors, we project the BVs onto the axis, which separates the primitives most.

In Figure 5.9b we move the direction of the first principal component on the x-axis. Now we compute the BV intervals $[S_i, E_i]$ and use the x-axis, more specifically the direction of the first component of the PCA, respectively, as sweep direction. Comparing Figure 5.9a with Figure 5.9b depicts the advantage of using the first principal component as sweep direction. The number of false-positives great reduces.

As a consequence, combining SaP and PCA reduces the number of primitive pairs tested for intersection and thus significantly reduces the calculation time.

Limitation

Using the first principal component as sweep direction only, will nevertheless produce false-positives, because of the dimensional reduction in the SaP step. The SaP technique, used to separate the primitives, projects all 3 D BVs to 1 D intervals. This means, for example, that in some cases primitives of the *front side* and primitives of the *backside* of an object will be recognized as potentially colliding pairs, even if there is a large distance between them (see Figure 5.10). This recognition will result in an amount of unwanted false-positives.

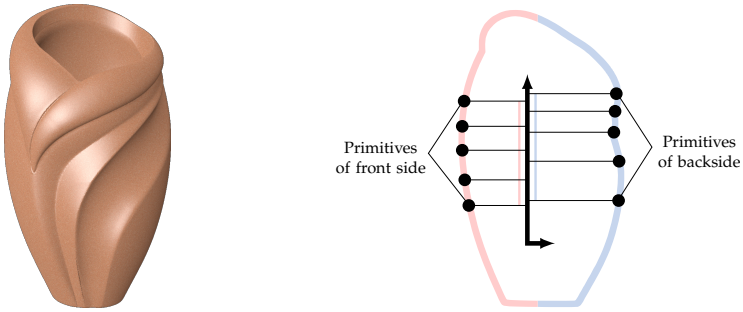


Figure 5.10: On the left: A 3 D model of a flower vase. On the right: The silhouette of the model with the corresponding PCA and the overlapping intervals of primitives of the front side and primitives of the backside.

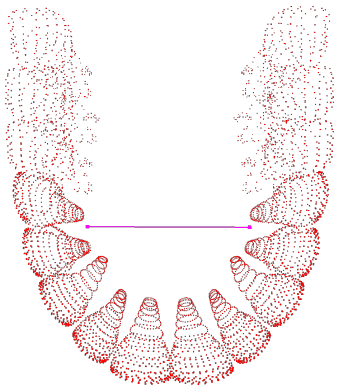
To eliminate this kind of false-positives we subdivide the scene into connected components using fuzzy c-means algorithm [Bez81; Ped05] (see Section 4.5).

5.2.2 *Principal Curves to Determine a Better Sweep Direction*

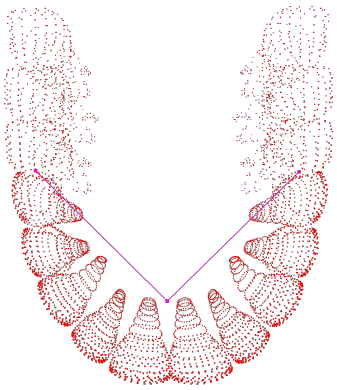
A further possibility for the determination of a sweep direction is the usage of principal curves. These nonlinear generalizations of principal components were first introduced by Hastie and Stuetzle [HS89] and Hattie [Hat84]. Hastie and Stuetzle [HS89] defined principal curves as "self-consistent" smooth curves, which pass through the "middle" of a d-dimensional distribution or data set. With this technique we can reduce the limitation of using the PCA as sweep direction (see Section 5.2.1).

Kégl, Krzyzak, Linder, and Zeger [Kég+00] presented a new algorithm, which represents a principal curve as a polygonal line. This approach is more robust than the Hastie-Stuetzle algorithm [HS89] because it uses a better heuristic to adapt smoothing term to fit the line segments to the data points. This approach tries to minimize the average distance from the curve rather than from the vertices of the curve (this is different from the other algorithms based on vector quantization, such as the Self-Organizing Map (SOM) [Koh90] or the generative topographic mapping). Furthermore, their approach is

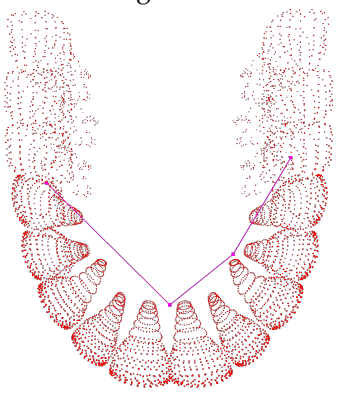
much faster than the Hastie-Stuetzle algorithm, particularly for huge data sets [Kég99; Kég+00].



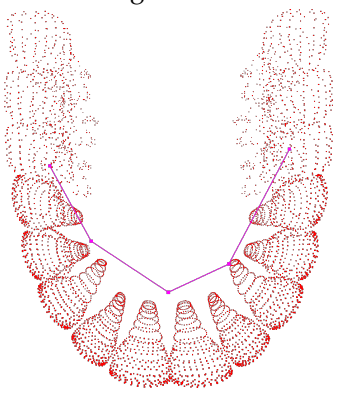
(a) A principal curve, consists of one line segment.



(b) A principal curve, consists of two line segments.



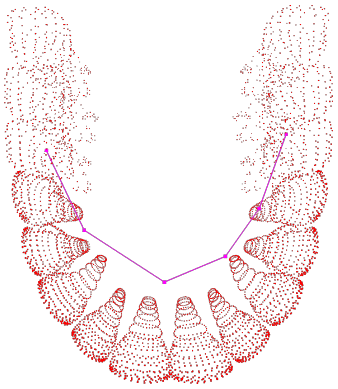
(c) A principal curve, consists of three line segments.



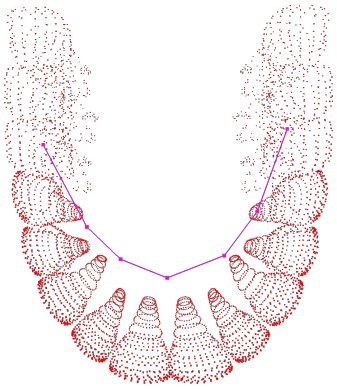
(d) A principal curve, consists of four line segments.

We implemented the approach presented by Kégl, Krzyzak, Linder, and Zeger [Kég+00] in C++ and CUDA. For a number of data points $x_i = x_1, \ldots x_n \subset \mathbb{R}$ this algorithm tries to find a polygonal curve with k segments and a predefined length. Figure 5.11 shows the functioning of the algorithm. In the first step a straight line segment (a part of the first PCA line) is used to fit the data points best (see Figure 5.11a). In the following steps the algorithm increases the number of line segments (in the outer loop, see Algorithm 5.6) by adding a vertex to the polygonal line (see Figure 5.11b-5.11h). If a new vertex is added to the principle curve, the position of all vertices is updated in the inner loop of the algorithm. The algorithm determines, if the number of line segments exceeds a predefined number.

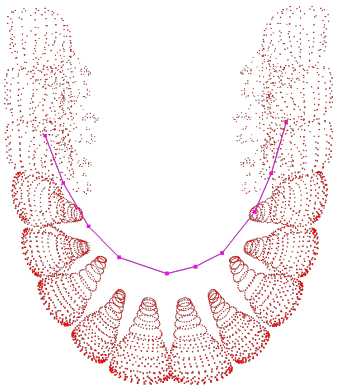
The inner loop of the algorithm (see Algorithm 5.6) is divided into two steps, a projection step and an optimization step. In the projection step all data points are subdivided into regions, so-called "nearest neighbor regions", depending on to which segment or vertex they are projected. In the optimization step the position of each principle curve vertex is updated by a line search to minimize an objective function. This objective function consists of an average squared distance



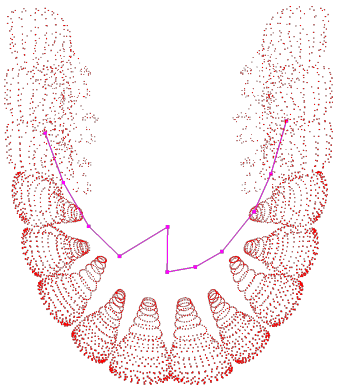
(e) A principal curve, consists of five line segments.



(f) A principal curve, consists of six line segments.



(g) A principal curve, consists of nine line segments.



(h) A principal curve, consists of ten line segments.

Figure 5.11: A 3 D model of the upper human denture with the corresponding principle curve containing of different numbers of segments.

term and a curvature penalty. These two steps are repeated till this procedure converges.

Figure 5.11g shows that the principle curve, created with this algorithm, passes through our 3 D data points. Using this curve as sweep direction for the SaP algorithm will separate the data points best. It is nevertheless noticeable that choosing to many line segments will produce a degenerated principle curve (see Figure 5.11h). The next step would be to straighten this curve, so that the resulting line can be used as a sweep direction directly. Therefore, the 3 D model must be deformed in the same way the principle curve is deformed. Here one must insure, that no collisions are added or removed, while deforming the model.

Complexity

The complexity of the inner loop is controlled by the complexity of the projection step, that is in $\mathcal{O}(n \cdot k)$, where n is the number of data

Algorithm 5.6 Principal Curve

```
Input: x_i data points
  initialization
  while #LineSegements < threshold do          ▷ outer loop
    repeat                                     ▷ inner loop
      projection
      vertex optimization
    until update objective function has converged
    add new line segment
  end while
```

points and k the number of line segments. The complexity for the whole algorithm, if we add one segment at a time, is in $\mathcal{O}(n \cdot k^2)$. Using a good stop condition [Kég99, Section 5.1.1] will reduce the computational complexity of the algorithm to $\mathcal{O}(n^{\frac{5}{3}})$. Furthermore, the complexity can be dramatically decreased if you add more than just one vertex. Adding a new vertex at the midpoint of every segment will reduce the computational complexity to $\mathcal{O}(n \cdot k \cdot \log k)$.

Conclusion and Future Work

Our current implementation is not yet fast enough to outperform the PCA approach. Therefore, the principle curve algorithm has to be parallelized perfectly—which is nearly impossible with the current approach—or a new algorithm to determine the principle curve in parallel is needed.

Furthermore, the deformation processes of the model to convert the principle curve into a line present several challenges. To use the principle curve for the SaP technique, we have to transform the curve into a line, while we transform the scene the same way. During this transformation no collision should be added or removed within the scene. Deforming a closed object can result in degenerated primitives, which is unwanted in all cases. A useful solution for the deformation process can be skinning [DB13; LCF00; Met92; Met96] or skeleton animation [KJP02; RLN06], which are commonly used in character animation and deformation.

5.2.3 *Implementation*

After the subdivision process the following steps (see Algorithm 5.7) can be performed for each cluster independently and therefore, i. e., each cluster can be distributed to a different GPU or can be processed in parallel on the same GPU (CUDA concurrent kernel execution).

In the first step, see Algorithm 5.7, we compute the PCA using the center of mass as data points of each primitive assigned to this cluster. To parallelize this computation we use some high performance functions from the Thrust library[1] [BH11]. To be more specific, we

1 http://thrust.github.com/

Algorithm 5.7 Sweep-Plane Technique Using PCA

Input: list L of primitives of all objects, number of cluster c
Output: colliding primitives
for all clusters **do in parallel**
compute PCA
transform primitives of cluster
↳ direction of first component of PCA points along x-axis
for all primitives assinged to this cluster **do in parallel**
compute Bounding Volume (AABB)
end for
sort BVs along x-axis
determine start and end position of each BV along x-axis
determine memory usage and number of threads per BV
create possible colliding pair list
for all pairs in possible colliding pair list **do in parallel**
perform overlap test on y-axis
if BVs don't overlap on y-dimension **then**
remove pair from possible colliding pair list
end if
end for
for all pairs (i_0, i_1) in possible colliding pair list **do in parallel**
if i_0 and i_1 share an edge **then**
return no collision
end if
if i_0 and i_1 intersect **then**
return collision
end if
return no collision
end for
end for

use `thrust::transform_reduce(...)` and `thrust::reduce(...)` to determine the covariance matrix. For the singular value decomposition we use `NR::svd(...)` from Numerical Recipes in C++ [Pre+07].

After the PCA computation process we perform a transformation step in the way that the direction of the first component of the PCA points along the x-axis of the original coordinate system. Now we compute the BVs for the transformed primitives (see Figure 5.9) in parallel. We initiate for each primitive its own CUDA thread. Due to the fact that we use Structure of Arrays (SoA) instead of Array of Structures (AoS) for all data structures (coordinates of the primitives, minimum and maximum coordinate values along each axis of each Bounding Volume, and more), we can access the memory in completely coalesced way (see Figure 3.3 for different types of memory access pattern).

For the SaP step we have to sort the array containing all start (S_i) and end (E_i) points of the BV intervals along the x-axis (we use an array per dimension). For this purpose, we use the very fast parallel sorting implementation from the Thrust library `thrust::sort_by_key(...)`. We have to perform a key-value sort because we use a tuple for the BV representation, which contains of the BV minimum or maximum value as key and an identifier as data value. The identifier

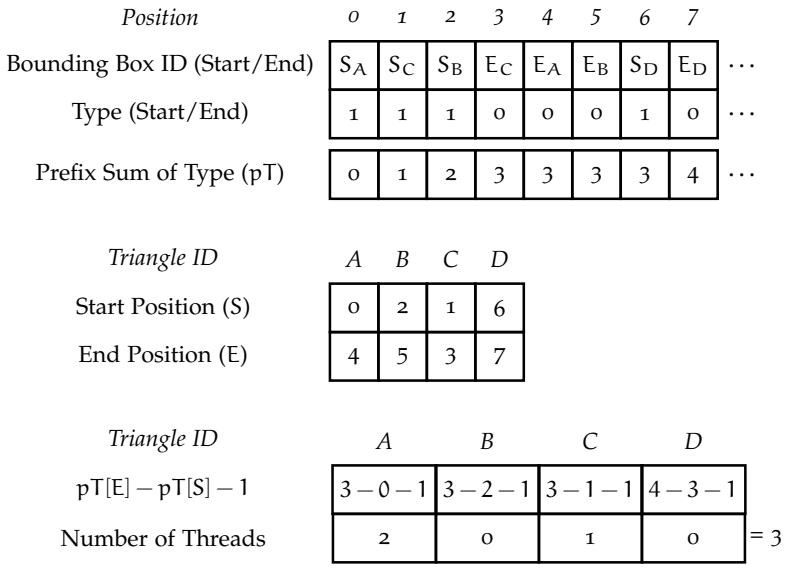


Figure 5.12: Determination of the minimal number of threads needed to identify all possible colliding primitive pairs and worst-case memory usage to store all these pairs.

states whether the key value is a minimum or maximum value and to which primitive the BV belongs.

5.2.4 *Thread Management*

In this section we depict how we determine the minimal number of working (CUDA) threads, which are needed to identify all possible colliding pairs. Furthermore, we compute the worst-case memory usage, i. e., the space needed to store all possible colliding primitives, at the same time. Additionally, an array "Type" with the information if at position j is a start ($S_j \rightarrow \text{Type} == 1$) or an end ($E_j \rightarrow \text{Type} == 0$) point is created during the BV creation process (see Figure 5.12 upper part).

On account of the fact that we want to avoid counting overlapping BVs twice, we only consider the start point (S_i) of a BV interval i. If this is not taken into account, and we consider both the start (S_i) and end point (E_i) of the BV intervals, for example in the case of $[S_a, S_b, E_a, E_b]$, we will receive two intersections. Primitive a intersects with primitive b, and vice versa. So, when we consider the start point (S_i) solely, we will get an intersection between primitive a and b only, because S_b is in the interval $[S_a, E_a]$, whilst S_a is not in the interval $[S_b, E_b]$.

To identify the number of working threads needed to do all intersection tests for a primitive, we need the amount of BV intersections between the BV of a primitive and all other BVs for all primitives. Therefore, a very suitable solution is the prefix sum algorithm from

the Thrust library `thrust::exclusive_scan(...)` using the "Type" array as input (see Figure 5.12 upper part). The resulting array pT can be used to compute the working threads needed for a primitive to do all possible intersection tests. Therefore, we calculate $pT[E_i] - pT[S_i] - 1$ for a primitive i, which generates the number of threads needed for the corresponding primitive i. The total amount of threads is equal to the number of the worst-case memory usage, required to store all possible colliding primitive pairs.

$$\#\text{threads} = \sum_{i=0}^{n-1} (pT[E_i] - pT[S_i] - 1) \tag{5.4}$$

where n is the number of primitives assigned to this cluster.

Going back to Algorithm 5.7 shows that we have to execute—depending on the scene—a huge amount of threads to create the possible colliding pair list. The accurate number of threads we need can be determined with Eq. (5.4). Each thread takes a possible colliding pair $P(p_i, p_j)$ and load the BV values of the y-dimension. Now an overlap test for the y-dimension is performed. If the BVs do not overlap in the y-dimension the possible colliding pair $P(p_i, p_j)$ is removed from the list. This overlap test will remove false-positives again. The last step of Algorithm 5.7 will check for all remaining possible colliding pairs in the list if both primitives sharing an edge or not and if that is not the case we perform an exact primitive-primitive intersection test. The result of the Algorithm 5.7 is a list of all intersecting primitives for this cluster.

5.3 FAST TRIANGLE-TRIANGLE INTERSECTION TEST

For our last step in our collision detection approach, the real primitive-primitive intersection test (in our case the primitives are triangles), we use the *interval overlap method* suggested by Held [Hel97] and Möller [Möl97]. In a first step this approach tests if the two face (triangle) normals can be used as separating axes. Therefore, we test if the vertices of triangle t_A lie completely on one side of the plane of the other triangle t_B. If this is the case, no intersection occurs. If this is not the case, then the planes containing the triangles are intersecting. They intersect in a line L, $L(t) = P + (\mathbf{n}_A \times \mathbf{n}_B)$, where $\mathbf{n}_A$ is the normal of triangle t_A and $\mathbf{n}_B$ of triangle t_B. In addition, this line is intersecting both triangles. The next step is to determine the scalar intersection intervals between L and both triangles. If both scalar intervals overlap, then these triangles intersect. Algorithm 5.8 provides a summery of the triangle-triangle intersection test.

Algorithm 5.8 Triangle-Triangle Intersection Test by Möller

Input: two triangles t_A and t_B
Output: intersection/no intersection
  determine plane equation p_A of triangle t_A
  if vertices of t_B on same side of plane p_A **then**
    return no intersection
  end if

  determine plane equation p_B of triangle t_B
  if vertices of t_A on same side of plane p_B **then**
    return no intersection
  end if

  determine intersection line L of two planes p_A and p_B
  determine scalar intersection intervals for each triangle with line L

  if intersection intervals do not overlap **then**
    return no intersection
  end if

  return intersection

5.4 COLLISION DETECTION BASED ON FUZZY SCENE SUBDIVISION

Algorithm 5.9 provides a concise overview of our new collision detection approach. In the first step we have to determine the data points, which we use in the subdivision process later. This step, how we determine the data points, is described in detail in Section 5.1.1. For the subdivision process many approaches exist (see Chapter 4). We decide to use a clustering algorithm for the partitioning. This procedure is described in Section 5.1.2. The last part of our approach is to create a list of primitives, which are possible colliding, followed by an exact intersection test. For more details of these procedures we refer to Section 5.2 and 5.3.

Algorithm 5.9 Collision Detection Algorithm

Input: list L of primitives of all objects
Output: list of intersecting primitives
  for all primitives in L **do in parallel**
    compute data points
  end for
  for all data points **do in parallel**
    subdivide scene into parts
  end for
  for all subparts of the scene **do in parallel**
    create possible collision pair list
    perform exact collision detection test
  end for

Our approach is performed completely on the GPU and therefore, no memory transfer between the host and GPU is needed. It is important to avoid this memory transfer because of its high transmission costs. Furthermore, more and more computer systems own more

than one GPU. To make full use of this multi-GPU systems (e. g., GPUs are connected via Scalable Link Interface (SLI)[2] or CrossFireX[3] or since Intel introduced the Sandy Bridge micro-architecture, each Intel CPU contains a GPU also) new approaches should be able to use more than just one GPU. This has been taken into consideration as we developed our new approach. We subdivide the scene into overlapping parts. Now each part can be handled independently and therefore, each part can be investigated by a different GPU. Consequently, our approach is especially well-trailered for multi-GPU systems.

5.5 ACCURACY AND LIMITATIONS

Our collision detection algorithm will recognize all intersections between all primitives. Therefore, our approach performs BV intersection tests with all primitives of a cluster, to detect all colliding primitive pairs. However, in the case of significant differences in the size of the primitives, it could happen that a primitive is completely assigned to one cluster, but collides with a primitive which is completely assigned to an adjoining cluster. The reason for this is that our approach uses the centroid, which represents a primitive, for the clustering process. To prevent this, we have to decrease the threshold value (membership) in the clustering step. This results in a higher degree of overlap between adjoining clusters (see Figure 5.13). The size of the overlap has to be at least as large as the overall maximum distance from primitive's centroid to one of its vertices. Taking the example of a triangle as primitive will result in:

$$\max_{i=1,2,\ldots,n}\left(\max_{k=0,1,2}\left(\|C_i - vertex_{i,k}\|_2\right)\right) \tag{5.5}$$

From this follows one small restriction for our approach. The large overlap between clusters can affect the performance in some scenarios, because of a higher number of collision computations. This limitation can be avoided by virtually subdividing huge primitives. The virtual primitives are used for clustering and sorting instead of the initial primitives.

If the size of all primitives is more or less equal, then our algorithm chooses a membership value so that the overlap between adjoining clusters consists of exactly two primitives.

2 SLI is a technology developed by NVIDIA. This technique allows to link two or more video cards together.

3 The multi-GPU solution provided by Advanced Micro Devices (AMD). Previously known as CrossFire.

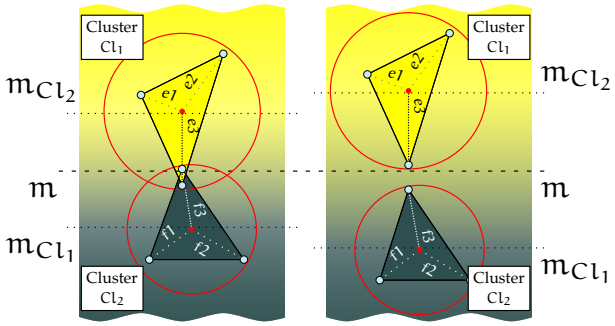


Figure 5.13: This Figure shows two adjoining clusters with two triangles as primitives, one colored in yellow and one in grey. The yellow triangle is completely assigned to the yellow cluster Cl_1 and the grey triangle is completely assigned to the grey cluster Cl_2. On the left side of the Figure we choose the overlap $d(m, m_{Cl_1}) == d(m, m_{Cl_2}) < \|f_3\|_2 < \|e_3\|_2$. Accordingly, like you can see in the Figure, it is possible that the yellow triangle intersect with the grey one. In this case, this collision will not be recognized by our collision detection. On the left side of the Figure we increase the overlap such that $d(m, m_{Cl_1}) == d(m, m_{Cl_2}) < \|f_i\|_2, i = 1, 2, 3$ and $d(m, m_{Cl_1}) == d(m, m_{Cl_2}) < \|e_i\|_2, i = 1, 2, 3$. As a result it is impossible that triangles, which are completely assigned to different clusters, can intersect.

5.6 BENCHMARK FOR DEFORMABLE OBJECTS

To evaluate the performance of our collision detection algorithm in different situations, we choose some often used collision detection benchmarks to compare our results against other approaches. It should be noted that these benchmarks only simulate a special case, where a huge amount of collisions and self-collisions occur. The most significant down side is that these benchmarks do not simulate all possible configurations between the colliding objects. In Section 5.7 we present a better Benchmarking Suite, but this Benchmarking Suite is so far not been able to handle deformable objects. However, an extended version with support for deformable objects is planned.

Experiments have shown that subdividing the scene into 2 respectively 4 clusters, when the objects are far apart from each other, for a single GPU provides the best performance. Therefore, in the following benchmarks we subdivided the scenes into 2 clusters.

5.6.1 *Implementation and System Details*

We have implemented our collision detection algorithm on a NVIDIA GeForce GTX 480 using the CUDA toolkit 5.0 as development environment. Because our collision detection algorithm is purely GPU-based, components like CPU and RAM do not have an effect on the

running time. However, for the sake of completeness, we will provide the key data of our system. Our collision detection algorithm is implemented in C++/CUDA. The platform for benchmarking consists of a PC running Gentoo Linux with an Intel Core i5-2500K 3.30 GHz CPU and 8 GB of memory. For sorting and prefix computation steps we used Thrust, a parallel algorithms library (shipped with the NVIDIA graphics card driver).

5.6.2 *Cloth on Ball Benchmark*

In this benchmark, a cloth, containing of 92 k triangles, drops down on a rotating ball, containing of 760 triangles (see Figure 5.14). Thereby the cloth has a huge number of self-collisions and many collisions between the cloth and the ball. This benchmark is discretized and therefore, subdivided into 93 frames. Our collision detection algorithm needs for this benchmark 20.24 ms in average (see Table 5.1).

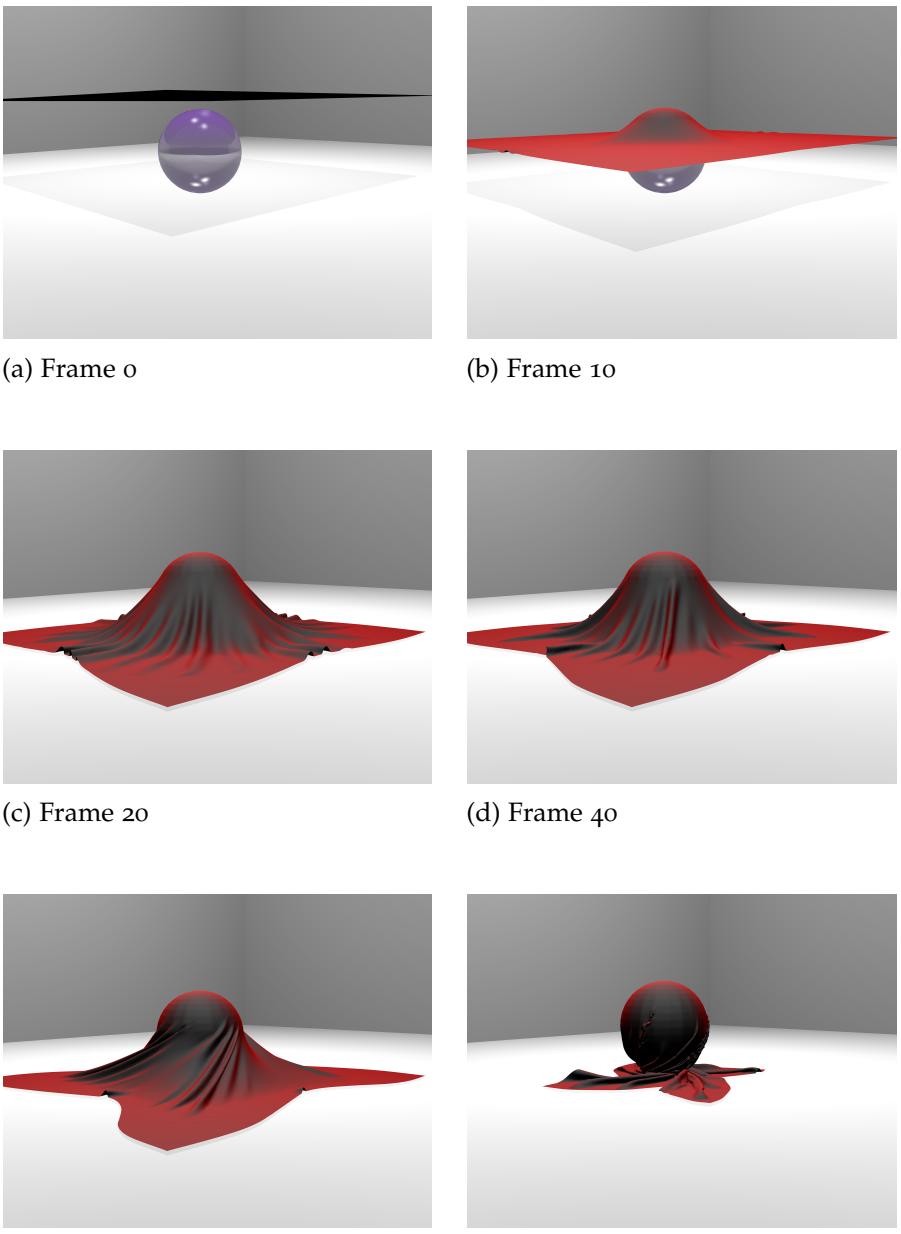
In Table 5.1 we show the average collision detection time needed for this benchmarks compared with state-of-the-art collision detection algorithms. Our approach is slightly slower than the CStreams [Tan+11] technique but this approach cannot be easily extended to more than one GPU. Comparing our approach to the hybrid CPU/GPU collision detection techniques [Kim+09; PKS10] and the multi-core collision detection approach [TMT10] shows that our technique performs better.

BENCH.	OUR	CST.	PAB.	HP	MC
Cl. on Ball	20.24	18.6	36.6	23.2	32.5

Table 5.1: Collision detection computation times in milliseconds. The timings include both external and self-collision detection. CStreams (CSt.) – GPU-based streaming algorithm for collision detection [Tan+11], Pab. – a hybrid CPU-GPU collision detection technique based on spatial subdivision [PKS10], HP – a hybrid CPU-GPU parallel continuous collision detection [Kim+09], MC – a multi-core collision detection algorithm running on a 16 core PC [TMT10].

Figure 5.15 shows that the collision detection time needed to compute all collisions from frame 60 onwards increase because the number of self-collisions increase heavily like you can see on the Figures 5.14e–5.14f. Our collision detection algorithm needs more time to collect all possible colliding triangles and has to do more intersection tests between them. The benchmark, provided by the UNC Dynamic Scene Benchmarks collection, itself contains intersecting triangles, which means that real collisions occur, like you can see at frame 93. It should be noted, that intersecting primitives should not

occur in such a benchmark because in a real-world scenario these objects would not interpenetrate.



(a) Frame 0 (b) Frame 10

(c) Frame 20 (d) Frame 40

(e) Frame 60 (f) Frame 93

Figure 5.14: Cloth on Ball simulation benchmark is a courtesy of the UNC Dynamic Scene Benchmarks collection and was provided by Naga Govindaraju, Ilknur Kabul, and Stephane Redon.

Figure 5.15 shows that the clustering and PCA computation step needs nearly in all frames the same time. This is due to the fact that we limit the maximum number of iterations for clustering process and thus, this step never takes too much time. Time needed for the AABB computation step can be regarded as negligible, like you can see in the timing. This step needs nearly no computation time. The

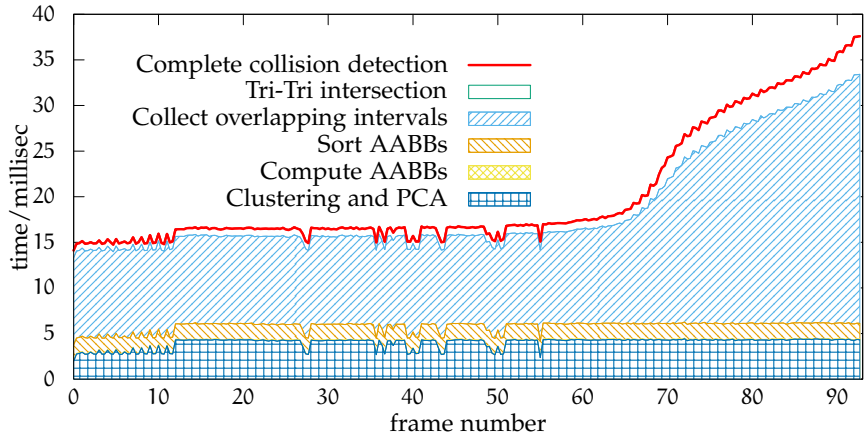


Figure 5.15: Collision detection time needed for Cloth on Ball Benchmark.

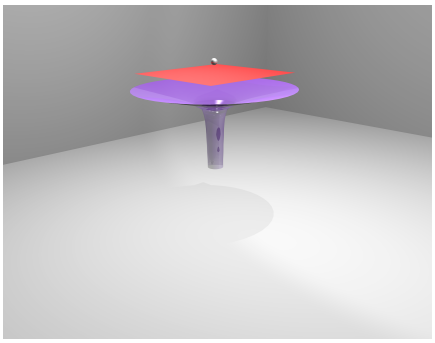
sorting step is nearly constant too, because we use a massively parallel sorting algorithm from the Thrust library. For this sorting method the order of the input data is equally, it needs the same time in any case. The most time consuming part is the step to create the possible colliding pair list. From frame 63 onwards the number of collisions increase heavily and therefore, the number of overlapping BVs. These BVs need to be further investigated which increase the overall computation time. Furthermore, more primitive-primitive intersection tests have to be done, because the number of touching primitives or primitives, which are very close together, increase.

5.6.3 *Funnel Benchmark*

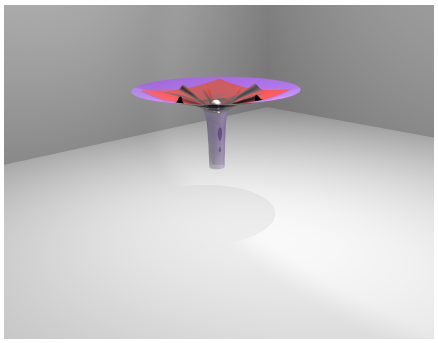
In this benchmark a flexible cloth, containing of 14.4 k triangles, falls into a funnel, containing of 2 k triangles, and passes through it, due to the force applied by a ball, containing of 1.7 k triangles. The ball slowly increased in volume over the time (see Figure 5.16). The whole benchmark is subdivided into 500 frames to provide a dynamic scene, where all objects can move and deform from one frame to another. This benchmark simulates situations with a high number of self-collisions and huge contact areas.

For this benchmark our collision detection algorithm needs 6.53 ms in average. In Table 5.2 we show the average collision detection time needed for this benchmarks compared with state-of-the-art collision detection algorithms. In this benchmark again, our approach is slightly slower than the CStreams [Tan+11] technique.

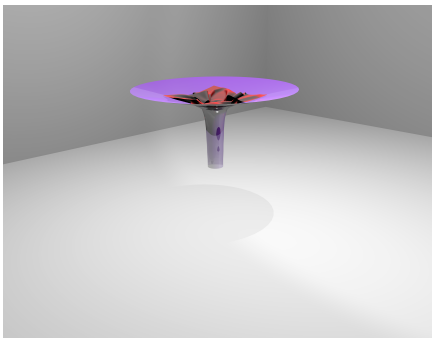
Figure 5.17 depicts that the collision detection time needed to compute all collisions increase slightly between frame 150 and frame 345. In these frames the cloth hits the funnel and slides a little bit into the funnel. From frame 345 onwards the ball pushes the cloth trough the



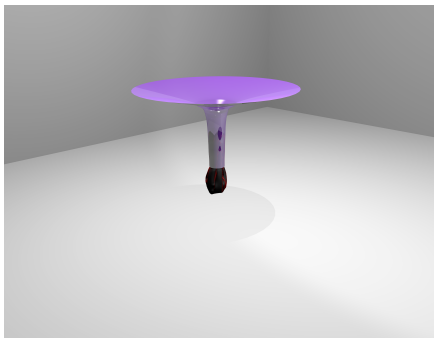
(a) Frame 1



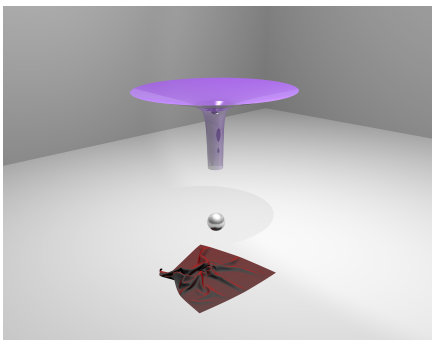
(b) Frame 125



(c) Frame 200



(d) Frame 375



(e) Frame 500

Figure 5.16: Funnel simulation benchmark is a courtesy of the UNC Dynamic Scene Benchmarks collection and was provided by Simon Pabst.

BENCH.	OUR	CST.	PAB.
Funnel	6.53	4.4	6.7

Table 5.2: Collision detection computation times in milliseconds. The timings include both external and self-collision detection. CStreams (CSt.) – GPU-based streaming algorithm for collision detection [Tan+11], Pab. – a hybrid CPU-GPU collision detection technique based on spatial subdivision [PKS10].

funnel, and produces a huge number of self-collisions, which results in a higher computation time needed for collision detection.

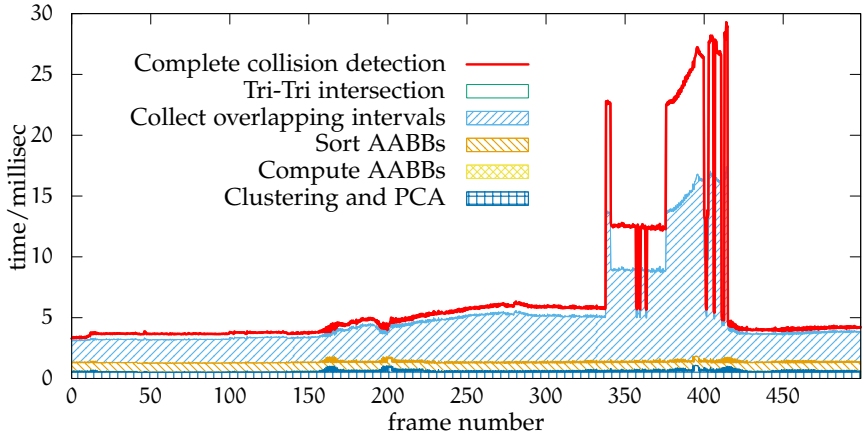


Figure 5.17: Collision detection time needed for Funnel benchmark.

This benchmark also shows that our approach needs the most time for the creation process of the possible colliding primitive pair list and the primitive-primitive intersection tests. Clustering, AABB computation and the sorting step are running in nearly constant time and take less than a half of the overall computation time.

5.7 EXCURSUS: OUR NEW BENCHMARKING SUITE FOR RIGID OBJECTS

In this section we like to briefly address our Benchmarking Suite [TWZ07; Wel+10] for rigid object collision detection and collision response schemes. Our benchmark suite[4] is published as open source and therefore, it is a great asset to users who want to figure out the best suited collision handling scheme to meet their specific requirements. Also researchers who want to compare their algorithms with other approaches using a standardized benchmark that delivers verifiable results can profit from our Benchmarking Suite.

5.7.1 *Overview of the Benchmarking Suite*

Our Benchmarking Suite is subdivided into two parts, like most other approaches too, a *Performance Benchmark* and a *Force and Torque Quality Benchmark*. With these benchmarks we can compare the time needed for a collision detection process and the quality of the computed force and torque.

4 http://cgvr.cs.uni-bremen.de/research/colldet_benchmark/

5.7.2 *Performance Benchmark*

Scenarios

Since this benchmark focuses on performance a Boolean collision responds (see Section 2.1.1) should be returned by the used collision detection approaches. In this case the collision algorithm does not need to collect all colliding points, it can stop on the first detected intersection.

- *Scenario I*: This scenario simulates situations where objects are in close proximity, but not touching. Most collision detection approaches use BVH and the worst-case for these algorithm is the case where primitives are very close but do not intersect and therefore, the leaves of the BVH collide even in deep levels in the hierarchy, but actually no polygon collision occurs. Consequently, the focus in this scenario is to place objects in a close proximity, but without an intersection.
- *Scenario II*: This scenario simulates situations where objects intersect. In some situations objects penetrate each other slightly and therefore, this scenario compares the time needed to determine the intersection.

A *configuration* describes the relative position and orientation between two objects. For rigid bodies such a configuration consists of 6 parameters shown in Figure 5.18: the transformation of object B in the coordinate system of object A, defined by the distance d, the polar coordinates φ_A and θ_A, and the rotation of object B, defined by the angles φ_B, θ_B, ψ_B.

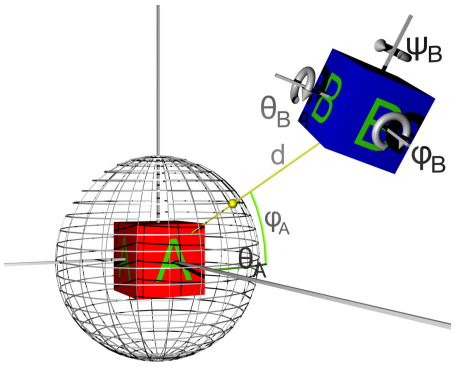


Figure 5.18: The relative position and orientation between two objects A and B, defined by 6 parameters.

The Grid Method

The first method uses a simple axis-aligned grid to find the translations. The midpoint of the moving object is positioned on the center

of all grid cells. In the following steps, the object is moving towards the fixed positioned object until a predefined distance is reached. If the predefined distance is reached, the configuration is stored. A downside of this approach is that the number of configurations found by this method is unknown in advance [TWZ07].

The Sphere Method

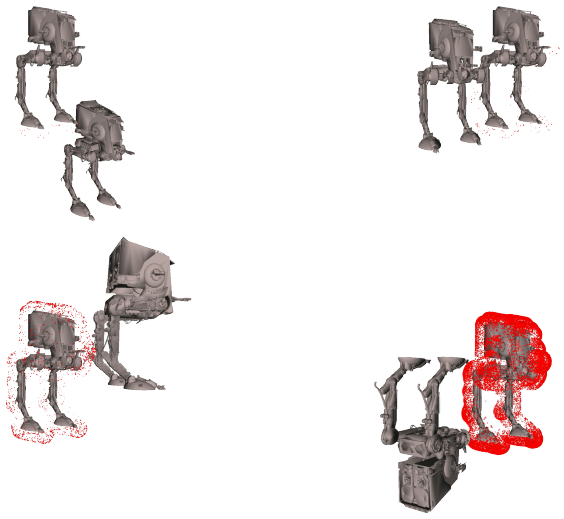


Figure 5.19: The sphere-method uses a fixed rotation for every cycle. An object is fixed and the moving object is rotated around. The rotation of the moving object changes after every cycle.

This method tries to reduce the time for finding possible configurations for a given distance. Therefore, the 3 D search space is reduced to two dimensions by using polar coordinates. It has to be mentioned that, this method can miss some interesting configurations. An object is placed in the middle on a sphere, while the moving object is placed on the sphere. The size of the sphere placed around the fixed object has to be at least as big as the fixed object plus the required distance. The next step we move the object on a straight line through the center of the sphere until we reach the required distance (Scenario I) or penetration depth (Scenario II), respectively (see Figure 5.19) [TWZ07].

After performing this procedure, we got a huge amount of configurations for predefined number of distances or penetration depth for an object-object pair. This computation process is a preprocessing step and has to be done only once, even if we change or add a new collision detection algorithm or change the underlying platform [TWZ07].

Scenario I

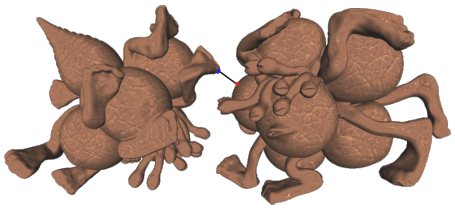


Figure 5.20: The minimal distance between two objects.

In order to calculate the distance d between two objects we need the two closest points from object A and object B (see Figure 5.20). So as to be scale-invariant, the distance is given in percent of the whole AABB of object A.

Object A has a fixed position and object B is placed on a sphere around object A. This sphere must be bigger than the AABB of object A plus the given distance d between both objects. In the next step, we move object B on a straight line to the center of object A until we reach the required distance or intersection volume. This configuration is then stored.

Our search space has 6 dimensions. To get as many configurations as possible consequently the configuration space must be sampled densely. For Scenario I we chose a step size of 15° for the spherical coordinates and a step size of 15° per axis for the rotations of object B. With these values, we generated a set of 1 991 808 sample configurations for each distance.

We computed sample configurations for distances from 0 up to 30 % of the object's BV size in 1 % steps, because in all example cases, there was no significant time spent on collision detection for larger distances. To compute the configuration of two objects with the correct distance we used Proximity Query Package (PQP) [GLM96; Lar+99].

Scenario II

In this scenario we use IST [WZ09] data structure to compute the configurations. A tetrahedron-based approach could not be used because of the probability relatively large computation times. Although ISTs compute intersection volumes very quickly, we still had to reduce the sampling of the configuration space. Therefore, we changed the step size per axis to 30°. We computed sample configurations for intersection volumes from 0 up to 10 % of the total fixed object volume in 1 % steps. With these values, we generated a set of 268 128 sample configurations for every intersection volume. Because most applications of collision detection try to avoid collision/intersection, an intersection volume of 10 % seems more than enough, as shown in Figure 5.21.

Figure 5.21: The intersection volume between two Happy Buddha statues.

5.7.3 *Force and Torque Quality Benchmark*

Our novel quality benchmark evaluates the deviation of the magnitude and direction of the virtual forces and torques ideal prediction models. Ideal forces and torques will be denoted by $\mathbf{F}^i$ and $\mathbf{T}^i$, respectively, while the ones computed by one of the collision detection algorithms—measured forces—will be denoted by $\mathbf{F}^m$ and $\mathbf{T}^m$.

Consequently, the scenarios in this benchmark, including objects and paths, should be meet two requirements:

a) They should be simple enough so that we can provide a model;

b) They should be a suitable abstraction of the most common contact configurations in force feedback or physically-based simulations;

In the following sections we introduce the implemented scenarios and methodology in order to evaluate force and torque quality.

Benchmarking Scenarios

Figure 5.22 shows all scenarios with their parameters; they are explained in the following.

SCENARIO 1 (A,B): A cone is translated while colliding with a block, maintaining a constant penetration. The penetration we chose is $\delta = \frac{1}{3}H = \frac{2}{3}r$ and the length of the trajectory is $L + 2a$. Two situations have been differentiated in this scenario: (a) $h > \delta$ and (b) $h \to 0$, i. e., the block is a *thin* rectangle.

Ideally, only forces should appear and they should have only a component in the positive y direction. Moreover, these forces should be constant while the cone slides on the block. This scenario evaluates the behavior of algorithms with objects that have flat surfaces or sharp

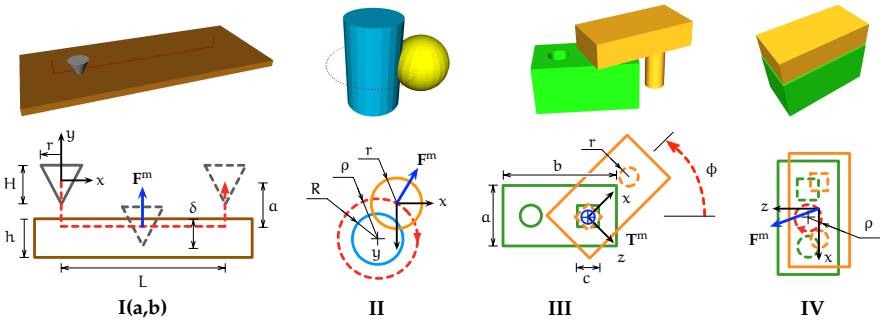


Figure 5.22: Scenarios in the Force and Torque Quality Benchmark, explained in Section 5.7.3. Upper row shows 3 D snapshots, whereas the lower displays parametrized schematics. Trajectories are represented with red dashed curves. Expected relevant forces and/or torques are shown with blue vectors. Coordinate systems are placed in points where forces and torques are measured – for the cone and the sphere this point is in their AABB center, whereas the position in z axis for the pins object is in the middle of the pin.

corners. In addition, Scenario Ib evaluates how algorithms handle the so-called *tunneling effect*, which occurs when thin or non-watertight objects yield too small forces and torques that allow interpenetration.

SCENARIO II: A sphere is revolved around a cylinder maintaining a constant penetration. The radius of the orbit is $\rho = \frac{5}{3}R = \frac{5}{3}r$. Ideally, only forces should appear (no torques) and they should have uniquely sinusoid components in x and y directions. In addition to that, the measured force magnitude should be constant while the sphere revolves around the cylinder. This is a suitable benchmark for environments with objects that have smooth, rounded surfaces.

SCENARIO III: A so-called *pins* object with a rectangular and a circular pin and a matching *holes* object compose this scenario. The rectangular pin is introduced in the rectangular hole and is turned around its axis. The size of the objects is $b = 2a$, the side of the rectangular pin is $c = 2r$ and it has a length of a in z direction. The maximum rotation angle is $\phi_{max} = 30°$. Ideally, only torques should appear and they should have only a component in positive z direction. Moreover, the measured torque magnitude should increase as ϕ increases. This scenario evaluates the behavior of algorithms with large contact areas.

SCENARIO IV: This scenario uses the same objects as in Scenario III. The start configuration is shown in Figure 5.22. Then, the pins object is revolved around the central axis of the second one. The orbit radius is $\rho = \frac{1}{10}c = \frac{1}{20}r$. The expected forces and torques are those that bring the *pins* object towards the central axis, i. e., sinusoidal

forces on the xy plane and torques with only z component. This scenario evaluates the behavior of algorithms with large and *superfluous* contact areas that should not generate collision reactions, such as the contact between objects in the xy plane. Besides that, this scenario contains small displacements around a configuration in which two objects are in surface contact. These small displacements should generate the corresponding small forces that push the *pins* object back to the *only-surface-contact* configuration.

Evaluation Method

For each scenario, we measured and recorded the following values for each time stamp k.

1. Forces $\mathbf{F}_k^m$,
2. Torques $\mathbf{T}_k^m$,
3. Penalty values q_k^m and
4. Computation time t_k.

In order to assess these measured values, we have developed ideal models of the expected forces and torques (i). The directions of these force and torque vector models are displayed in Figure 5.22, whereas magnitudes are considered to be proportional to analytically derivable collision properties, such as

1. $\|\mathbf{F}^i\|$ or $\|\mathbf{T}^i\| \sim p$, translational penetration depth,
2. $\|\mathbf{F}^i\|$ or $\|\mathbf{T}^i\| \sim V$, intersection volume.

In each scenario, we have determined p and V, respectively, as follows:

- Scenario Ia: $p \sim \delta$ and $V \sim \delta^3$
- Scenario Ib: $p \sim \delta$
- Scenario II: $p = \rho = \text{const}$ and $V = \text{const}$
- Scenario III: $p \sim \sin\left(\frac{\phi}{2}\right) - 1$ and $V \sim \left(\frac{1}{\tan(\phi)} + \frac{1}{\tan\left(\frac{\pi}{2}-\phi\right)}\right)\left(\sqrt{2}\cdot \cos\left(\frac{\pi}{4}-\phi\right) - 1\right)^2$
- Scenario IV: $p = \rho = \text{const}$ and
  $V = c^2 - (c - \rho|\cos\phi|)\,(c - \rho|\sin\phi|) + \pi r^2 - 4\int_{\frac{\rho}{2}}^{r}\left(r^2 - \tau^2\right)\mathrm{d}\tau$

In order to evaluate the quality of the magnitude, the standard deviation of measured (m) and ideal (i) curves is computed:

$$\sigma_F = \frac{1}{N}\sqrt{\sum_{k=1}^{N}\left(\,\|\hat{\mathbf{F}}_k^i\| - \|\hat{\mathbf{F}}_k^m\|\,\right)^2}, \tag{5.6}$$

where $\hat{\mathbf{F}} = \frac{\mathbf{F}}{\|\mathbf{F}\|_{max}}$, and N being the total amount of time stamps.

Analogously, the indicator for direction deviation is the angle between ideal and measured values; the average value of this angle is:

$$\gamma_F = \frac{1}{N}\sum_{k=1}^{N}\arccos\frac{\mathbf{F}_k^i\cdot\mathbf{F}_k^m}{\|\mathbf{F}_k^i\|\cdot\|\mathbf{F}_k^m\|} \tag{5.7}$$

Deviation values for torques (σ_T,γ_T) are computed using $\mathbf{T}_k^m$ and $\mathbf{T}_k^i$, instead of force values.

Additionally, we measure the amount of noise in the measured signals. A color coded time-frequency diagram using short time Fourier transform can be used to visualize the noise in time domain. In order to define a more manageable value for evaluations, we compute the ratio

$$\nu = \frac{\int S^m}{\int S^i}, \tag{5.8}$$

where S^m is the energy spectral density of the measured variable (e.g. $\|\mathbf{F}^m\|$) and S^i is the spectrum of the corresponding ideal signal. ν can be evaluated for forces and torques directions and magnitudes separately.

Equivalent Optimized Resolutions for Comparing Different Algorithms

Usually, when the quality of resolution is improved, whereas computation time increases. Therefore, an appropriate trade-off between quality and time performance must be found.

When properly evaluating or comparing collision detection algorithms, such a resolution must be found that makes possible to compare algorithms' quality for a given average performance, or to compare their performance for a given desired quality. In this context, we name "equivalent" optimized resolutions such resolutions with which algorithms exhibit a same desired time performance, being possible to fairly compare their qualities.

Considering two objects in a scenario (A is dynamic, B is static), we define the resolution pair (e^A_{opt}, e^B_{opt}) to be the optimum *equivalent* resolution pair:

$$\left(e^A_{opt}, e^B_{opt}\right) = \min\left\{\eta\left(e^A, e^B\right) \mid \bar{t}\left(e^A, e^B\right) = \tau\right\}, \tag{5.9}$$

where τ is the maximum admissible average computation time, $\bar{t}$ and $\eta = \omega_\sigma \sigma + \omega_\gamma \gamma$, the equally weighted sum of the standard deviations.

In practice, since time and quality functions of Eq. (5.9) are unknown, performed evaluations were carried out numerically after running several tests. For each scenario and algorithm, we defined three different resolutions within a reasonable[5] domain for each object A and B, building sets of $3 \times 3 = 9$ pairs (e^A, e^B). Then, the sets of 9 corresponding tests were performed, recording all necessary average computation times ($\bar{t}$) and global deviations (η) in each one. Next, we applied a linear regression to values of $\bar{t}$, obtaining the plane which predicts the average computation time for a resolution pair in each scenario. Each of these planes was intersected with $\tau = 0.9\,\text{ms}$[6], obtaining the lines formed by all (e^A, e^B) expected to have $\bar{t} = 0.9\,\text{ms}$ for each scenario.

Being aware of the fact that further refinements would yet be possible, it is considered that the reached compromise is accurate enough in order to make a fair comparison. The average absolute difference between predicted and measured η values with *equivalent* resolutions was 1.2 % for the Voxmap-Pointshell (VPS) algorithm and 2.1 % for the IST algorithm.

5.7.4 *Results*

Benchmarking is not as time consuming as configuration computation. To perform the benchmark, we load the set of configurations for one object. For each object-object distance and intersection volume respectively, we start timing, set the transformation matrix of the moving object to all the configurations associated with that distance, and perform a collision test for each of them. After that, we get a maximum and an average collision detection time for the given distance or intersection volume, respectively. Overall, we did 65 million different collision detection tests with each collision detection library.

Configuration computation

To compute all these configurations we used a PC cluster with 25 cluster nodes, each with 4 Intel Xeon CPUs and 16 GB of RAM. The time needed to calculate configurations for a complete set of distances or intersection volumes varies from object to object between 10 h and 200 h. Overall, we computed configurations for 86 objects, which lasted 5 600 CPU days.

5 Between coarse but acceptable and too fine resolutions.

6 Collision detection and force computation must lie under 1 ms; hence, we chose a resonable value under this barrier.

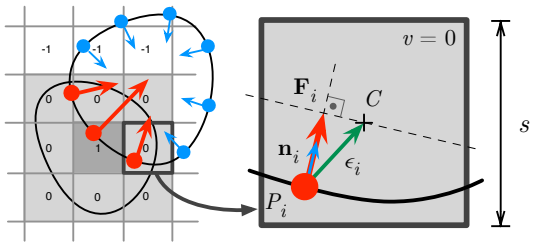


Figure 5.23: On the left: A layered voxmap (bottom) is colliding with red pointshell points (top), yielding red bold collision forces. On the right: The computation of a single collision force related to a colliding point is graphically shown. Single collision forces are computed scaling the normal vector ($\mathbf{n}_i$) of the colliding point (P_i) with the sum of the local ($\mathbf{n}_i \epsilon_i$) and global ($v \cdot s$) penetration of the point in the object.

Algorithms

In order to test our Benchmarking Suite, we used two collision detection algorithms, VPS and IST. Both algorithms use a penalty-based haptic rendering method, which allows colliding objects to penetrate each other to some degree. Each algorithm uses different penalty values: the one from VPS is the *penetration depth*, while the one from IST is the *intersection volume*.

First, we explain the algorithms and how they compute force and torque values they return to our benchmark. After this, we discuss the output of the Performance Benchmark and the Force and Torque Quality Benchmark.

THE VOXMAP-POINTSHELL ALGORITHM: The Voxmap-Pointshell (VPS) algorithm was initially presented by McNeely, Puterbaugh, and Troy [MPT99]. The algorithm computes collision forces and torques of potentially big and complex geometries with 1 kHz update rates. To achieve this goal, two types of data structures are generated offline for each colliding object-pair: a voxmap and a pointshell (see Figure 5.23). In this work, we used the fast and accurate voxmap generator presented by [Sag+08].

Voxmaps are 3 D grids in which each voxel stores a discrete distance value $v \in \mathbb{Z}$ to the surface. Pointshells are sets of points uniformly distributed on the surface of the object; each point has additionally an inwards pointing normal vector.

During collision detection, the normal vectors $\mathbf{n}_i$ of colliding points P_i—those which are in voxels with $v \geq 0$—are summed, after being weighted by their penetration in the voxmap, yielding the collision force $\mathbf{F}$. Torques $\mathbf{T}_i$ generated by colliding points are the cross product between forces $\mathbf{F}_i$ and point coordinates P_i, all magnitudes expressed in the pointshell frame, with its origin in with

being the center of mass. At the end, these torques $\mathbf{T}_i$ are summed to compute the total torque $\mathbf{T}$.

THE INNER SPHERE TREE ALGORITHM: Inner Sphere Tree (IST) [WZ09] is a novel geometric data structure, that provides hierarchical BVs from the *inside* of an object. The main idea is to fill the interior of the model with a set of non-overlapping spheres that approximate the object's volume closely. Therefore ISTs and, consequently, the collision detection algorithm are independent of the geometry complexity; they only depend on the approximation error.

The penetration volume corresponds to the water displacement of the overlapping parts of the objects and, thus, leads to a physically motivated and continuous repulsion force. The algorithm determines all pairs of overlapping spheres and computes a force for each of them. Summing all these pairwise forces gives the total penalty force $\mathbf{F}$. Similarly, the torque is computed separately for each pair of intersecting spheres and accumulated to obtain the total torque $\mathbf{T}$.

Discussion of the Benchmark Results

In this section we present the results returned from our benchmarks. The algorithms presented in Section 5.7.4 were used for this propose.

It is very hard to tell which algorithm is better because this is very dependent on the requirements. Our benchmarking provides a wide range of test cases to evaluate the given algorithm and return the computation time and the computed values. In the next sections we explain the results returned by the tested algorithm.

Figure 5.24: Some of the objects we used in our Performance Benchmark: A model of a Happy Buddha (1 087 716 polygons), a Chinese Dragon (1 311 956 polygons), a Circular Box (1 402 640 polygons) and a Gargoyle (1 726 420 polygons).

RESULTS OF THE PERFORMANCE BENCHMARK

Appart from the distance or the penetration depth between objects, the performance of the most collision detection libraries mainly depends on the complexity and the shape of the objects. Figure 5.24 shows some of the objects we used. All objects that are in the public

domain can be accessed on our website[7]. Within our benchmarks, we tested a model against a copy of itself. Of course, our benchmark also supports the use of two different objects, but the first method is sufficient to draw conclusions about the performance of the libraries.

We tested the libraries on an Intel Core2 CPU 6700 @ 2.66 GHz and 2 GB of RAM running Linux. All source codes were compiled with gcc 4.3.

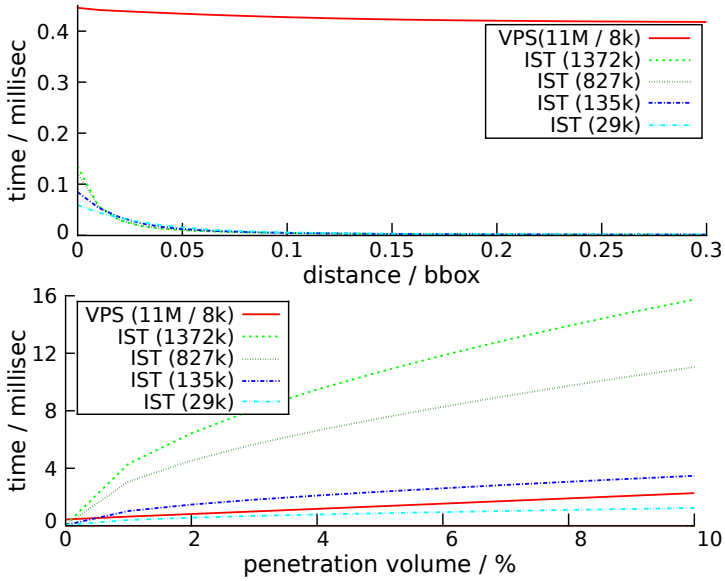


Figure 5.25: Performance Benchmark: Example result for Happy Buddha. The top plot shows the measured average collision response time for Scenario I (no collision) and the bottom one for Scenario II (collision) (see Section 5.7.2). Distance 0.0 means that the objects are touching. Volume 1 % means that the intersection volume is equal to 1 % of the total object volume. The number in parentheses after IST denotes the number of spheres (see Section 5.7.4). The two numbers after VPS denote the number of voxels and points, respectively (see Section 5.7.4).

An example of a result of the Performance Benchmark is shown in Figure 5.25, using Happy Buddha as object. Our Performance Benchmark facilitates a comparison of different algorithms as well as an assessment of the behavior of one algorithm, with respect to the objects complexity.

With the results from the Performance Benchmark it is now possible to compare collision libraries regarding their collision response time. These tests can also be used to determine objects or a placement of two objects which are not ideal for the tested algorithm. It is also possible to determin the influence of the object complexity or a lower approximation error on the collision response time.

7 http://cgvr.cs.uni-bremen.de/research/colldet_benchmark/index.shtml

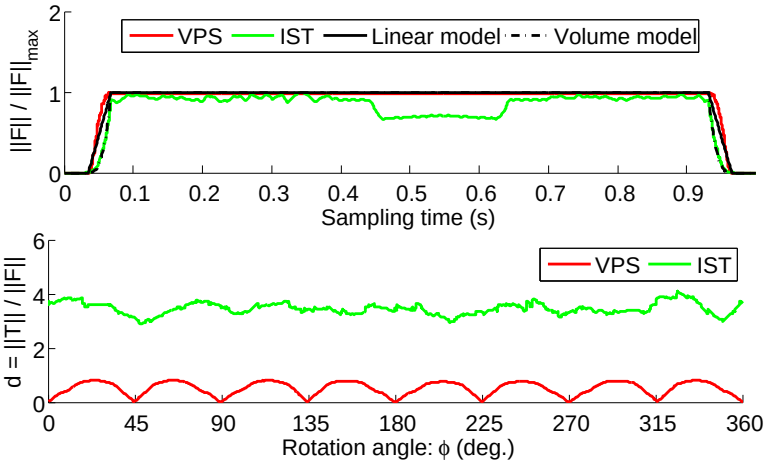


Figure 5.26: Force and Torque Quality Benchmark: On the top, an example for the normalized collision force vector computed by the tested algorithms (Scenario I) and on the bottom the orientation of the vectors (Scenario II).

However, the computation time is not enough to fully assess a collision detection algorithm. Often, the quality of the collision responses is another important factor. This is discussed in the next section.

RESULTS OF THE FORCE AND TORQUE QUALITY BENCHMARK

As in the case of the Performance Benchmark, all objects and paths used in the Force and Torque Quality Benchmark (see Figure 5.22) are available on our website[8]. We tested them on an Intel Core2Quad CPU Q9450 @ 2.66 GHz and 3.4 GB of RAM running Linux SLED 11. The libraries were compiled with gcc 4.3.

For the voxel size s we have chosen a fixed lenght unit u in the voxelized objects such that $H = 60u$, $h = 30u$ (Scenario I), $R = 30u$ (a penetration of $20u$ is maintained) (Scenario II), $c = 20u$ (Scenario III), and $\rho = 20u$ (Scenario IV). The number of voxels was chosen to be $728 \times 24 \times 303$ voxels for the block in Senario I while the cone has 15 669 pointshell points. In Scenario II, we used $491 \times 816 \times 491$ voxels for the cylinder and 12 640 pointshell points for the sphere. In Scenario III, the number of voxels was chosen to be $1\,204 \times 604 \times 603$ for the holes and 12 474 pointshell points for the pins object. For the last Scenario, the number of voxels was chosen to be $243 \times 123 \times 123$ for the holes and 13 295 pointshell points for the pins object.

Figures 5.26 and 5.27 show example plots of the magnitude analysis. The top of Figure 5.26 contains the expected model curves for ideal force magnitudes in Scenario I. Measured curves are superposed to expected curves to give an idea of how reliable they are derived with respect to these proposed collision response models. The standard deviation between measured and ideal curves yields the magnitude deviation $\sigma_F = 0.043$ for VPS and $\sigma_F = 0.176$ for ISTs.

8 http://cgvr.cs.uni-bremen.de/research/colldet_benchmark/index.shtml

In Scenario III, the standard deviation between measured and ideal curves yields the magnitude deviation $\sigma_T = 0.169$ and $\sigma_T = 0.112$ for the torques, respectively. The bottom of Figure 5.26 shows the curve $\frac{||T||}{||F||}$, which should be 0 for Scenario II, since ideally no torques should appear. This quotient gives information about the magnitude of forces or torques that actually should not occur.

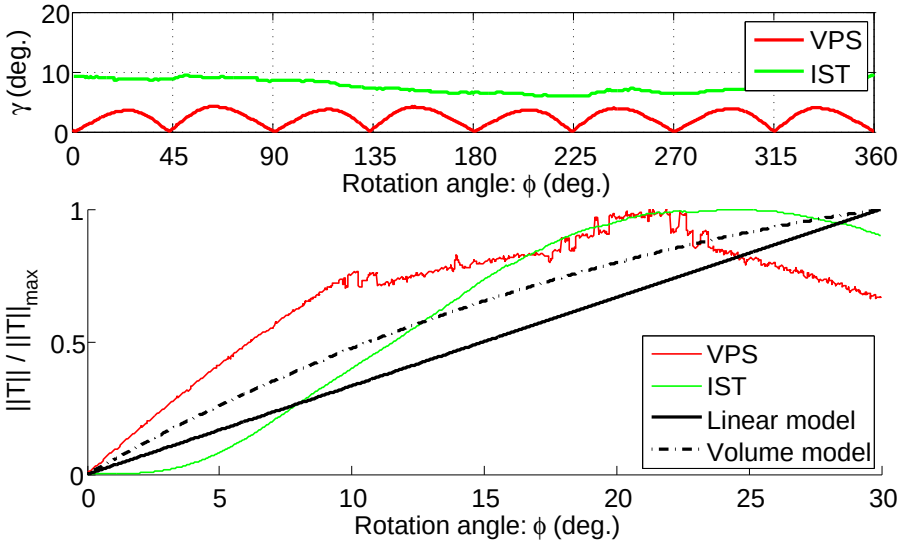


Figure 5.27: Force and Torque Quality Benchmark: On the top, an example for an average angle between model and measured forces (Scenario II) and on the bottom the normalized collision torque vector computed by the tested algorithms (Scenario III).

In Figures 5.28 and 5.29, force and torque components are displayed, giving a visual idea of force and torque direction deviations. The top plot of Figure 5.27 shows this direction deviation for Scenario II, the associated γ values are $\gamma_F = 2.40$ for VPS and $\gamma_F = 7.64$ for ISTs.

Finally, Figure 5.30 shows the results of our noise measurement of the force in the x-direction in Scenario III. The color coded time-frequency diagrams visualize the amount, the time, and the frequency of the signal's noise. The corresponding ν values are $\nu_F = 0.620$ for VPS and $\nu_F = 1.12$ for ISTs, where values closer to one denote a minor amount of noise.

All these results show that VPS and IST are very close to their underlying models and that different haptic rendering algorithms can be evaluated. From these results we can say that our models for penetration are suitable. Furthermore, they prove empirically that our benchmark is vaild. In particular, the benchmark also reveals significant differences between the algorithms: Whereas ISTs seem to have a higher standard deviation from the ideal model, VPS tends to deliver noisier signal quality. The decision between accuracy and noise could be essential for some applications.

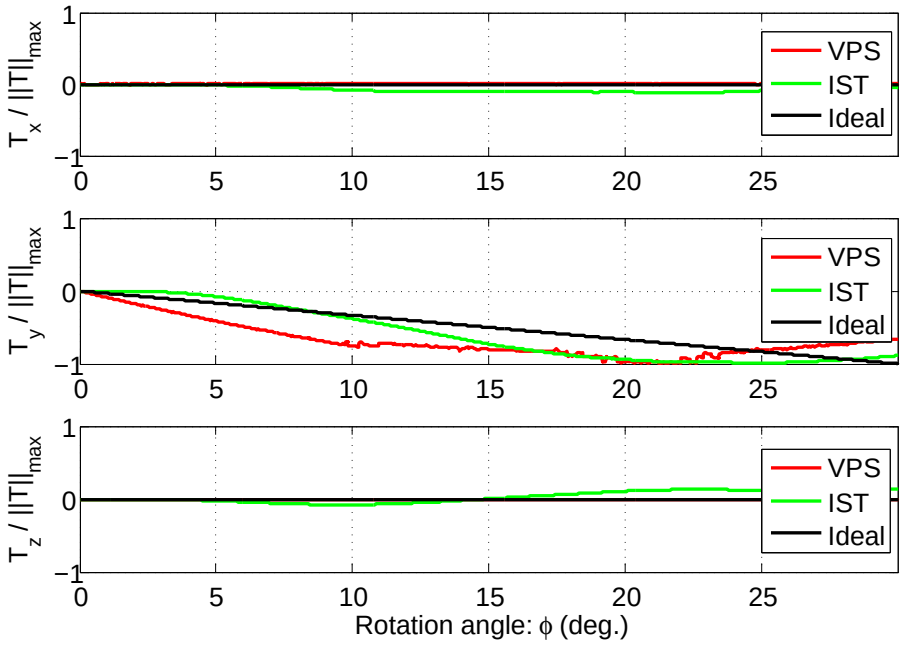


Figure 5.28: Force and Torque Quality Benchmark: An example for the collision torque (Scenario III) computed by the tested algorithms.

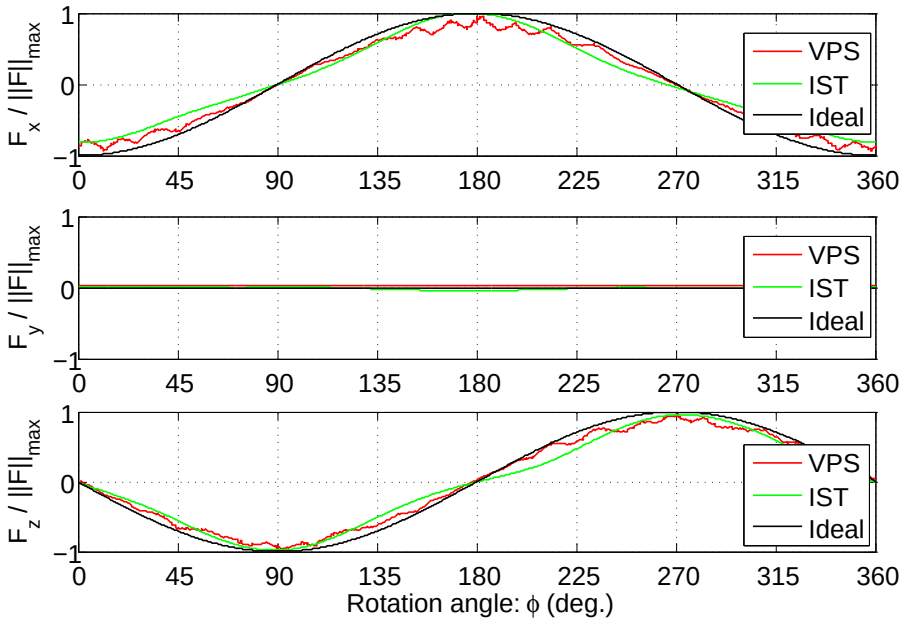


Figure 5.29: Force and Torque Quality Benchmark: An example for the collision force computed by the tested algorithms (Scenario IV).

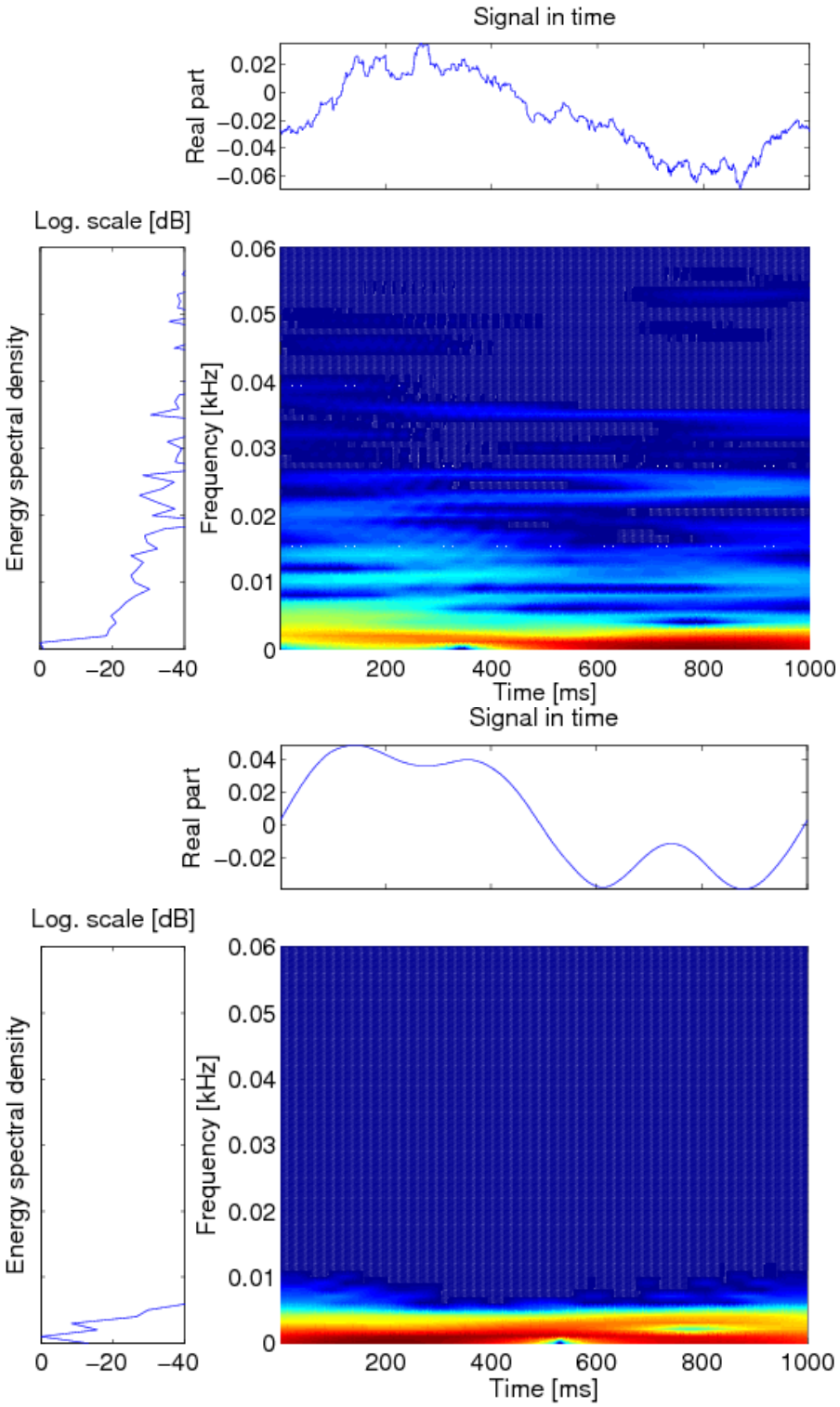


Figure 5.30: Force and Torque Quality Benchmark: On the top, the noise in the force signal of the VPS algorithm and on the bottom noise in force signal of IST algorithm. The colored picture shows the time frequency domain: The colors decode the intensity of the frequency, where dark blue represents an intensity of zero.

5.7.5 *Conclusions and Future Work*

The results maintain the validity of our analytically derived force and torque models. In addition, they show that quite different collision detection algorithms can be easily benchmarked with our proposed methods.

This Benchmarking Suite is published as open source, so it is a great asset to users who want to figure out the best suited collision handling scheme to meet their specific requirements, as well as to researchers who want to compare their algorithms with other approaches using a standardized benchmark that delivers verifiable results. Moreover, it helps to identify geometric cases in which the collision response scheme diverges from the correct results.

In the future, it would be nice to generate a ranking of the different measurements, like continuity of forces and torques in magnitude and direction or the noise of the signals, with respect to psychophysical cognition. To achieve that, elaborate user studies need to be done, including testbeds with different haptic devices and investigations about the perception of the different parameters. Another promising future project would be to extend our Benchmarking Suite for multibody-simulations.

Finally, a standardized Benchmarking Suite for deformable objects is still missing and would be very helpful for users and researchers. So far there does not exist a Benchmarking Suite for deformable models to compare the computed force and torques. Therefore, it is not possible to compare different approaches with respect to their quality. Currently all dynamic scenes are stored frame per frame in a separate file e. g., ply, obj, wrl, and so on. Thus, a user can load a scene file per file and perform a collision check. However, most commonly used benchmarks for deformable collision detections contain inaccuracies like real intersections, although there are unwanted in that scenario (see Figure 5.14). Therefore it will be essential to develop a robust and correct collision detection benchmark for deformable collision detection algorithm, that covers all cases that can occur in a deformation process. An interesting possibility is the usage of a skeleton representation for the object, like in skeleton animation [KJP02; RLN06]. Now the skeleton can be used to deform the object and to determine collision configurations. In this way self-collisions can be determined if an object is greatly modified by the skeleton. This approach is only possible if a skeleton representation can be determined. Furthermore, some objects can not be deformed in a natural way by a skeleton, e. g., a sphere. Another possibility is to move the vertices of mesh individually. This approach is the most flexible but this will lead to a huge amount of configurations.

5.8 FUTURE WORK

Nevertheless, our novel collision detection approach for soft bodies provides still room for improvements as regards the clustering process. Right now we are using fuzzy c-means for the subdivision process, but like we mentioned in Section 4.5 it is possible that this approach gets stuck in local optima. Therefore, we should use a robust clustering approach for the first partitioning step, like BNG. In the running time critical simulation process we use the very fast fuzzy c-means algorithm to update the clustering result. This step will make our approach more robust, so it will perform very fast in all situations.

Another improvement is to use a principle curve (see Section 5.2.2) instead of the direction of the first component of the PCA. A principle curve fits the shape of an object better than a straight line, which will reduce the number of false-positives. There should be much room for improvement because this step takes the most time in our collision detection approach.

Right now we use the Thrust sorting algorithm which does not make use of the temporal coherence that is inherent in most real scenes. Every frame the algorithm sorts the whole array from the beginning. This is, for example, the case if in the virtual scene from one frame to the next one some objects are moving, but the total order of the BVs along the sweep axis does not change that much. Therefore, an adaptive sorting approach will improve the sorting step and thus, reduce the collision detection time.

Distributing the computation process to more GPUs provides several additional problems. If we subdivide the scene into 2 clusters and we have two GPUs, each GPU can process one cluster. But in the case of subdividing the scene into 3 clusters, we have to decide which GPU has to process 2 clusters and which GPU has to process only one cluster. Assume a computer system has two different GPUs with widely varying computation performance, therefore, it could be best to subdivide the scene into 3 clusters and the GPU with highest computation power should process 2 clusters and the other GPU one cluster. Consequently, our collision detection approach needs a good distribution system to distribute the computation processes.

6 TECHNICAL DETAILS AND APPLICATIONS

This chapter gives a more technical view of our novel collision detection algorithm. We provide the data throughput through each part of our collision detection approach. Additionally, we show a sequence diagram, which illustrates the most important object interactions arranged in time sequence. Furthermore, to proof the functionality of our collision detection implementation we have implemented them into a commonly used real-time physics simulation called Bullet.

6.1 DATA FLOW

Let us assume that our collision detection is integrated into a physics simulation. In the first step the physics simulation calculates the new position for each primitive used in the simulation by applying all determined forces e. g., acceleration, velocity, gravity, and so on. The new physics entity—in our case a triangle mesh representation of the whole scene—is forwarded to our collision detection (see Figure 6.1). If the simulation is not running or the data are not already on the GPU, all mesh data points have to be transferred into GPU memory.

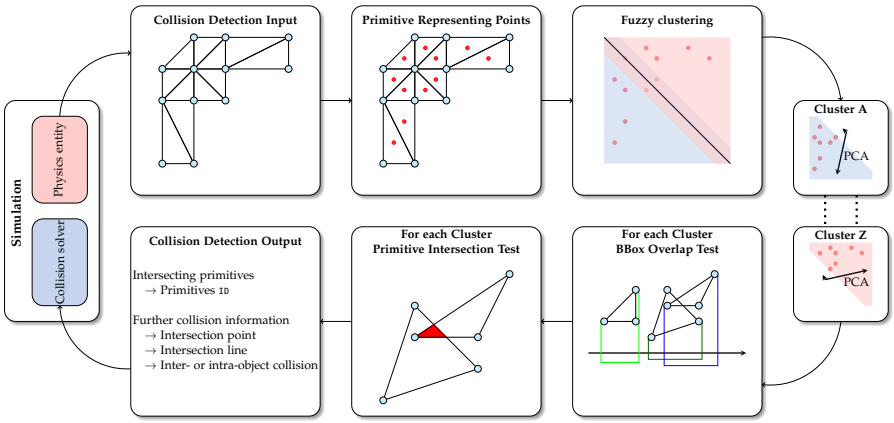


Figure 6.1: Data flow through our novel collision detection algorithm. Note that we show only the most important parts.

Our collision detection agreed upon additional information about the virtual scene such as primitive representation points, clustering information, BVs and more. Therefore, our approach performs several steps, e. g., clustering process, Sweep-and-Prune, and exact intersection tests. All additional data values are stored temporarily on the GPU. The result of a single collision detection step—all intersecting primitives—is sent back to the simulation. In the following the simu-

lation has to resolve the collisions between all reported primitives in a physical correct or plausible way.

6.2 SEQUENCE DIAGRAM

The sequence diagram (see Figure 6.2) shows main parts of our collision detection approach and the way they interact. Furthermore, the time flow of the entire process is specified by the sequence diagram. The activation boxes (light blue vertical rectangles) denote that the corresponding part of our collision detection is currently processing a request.

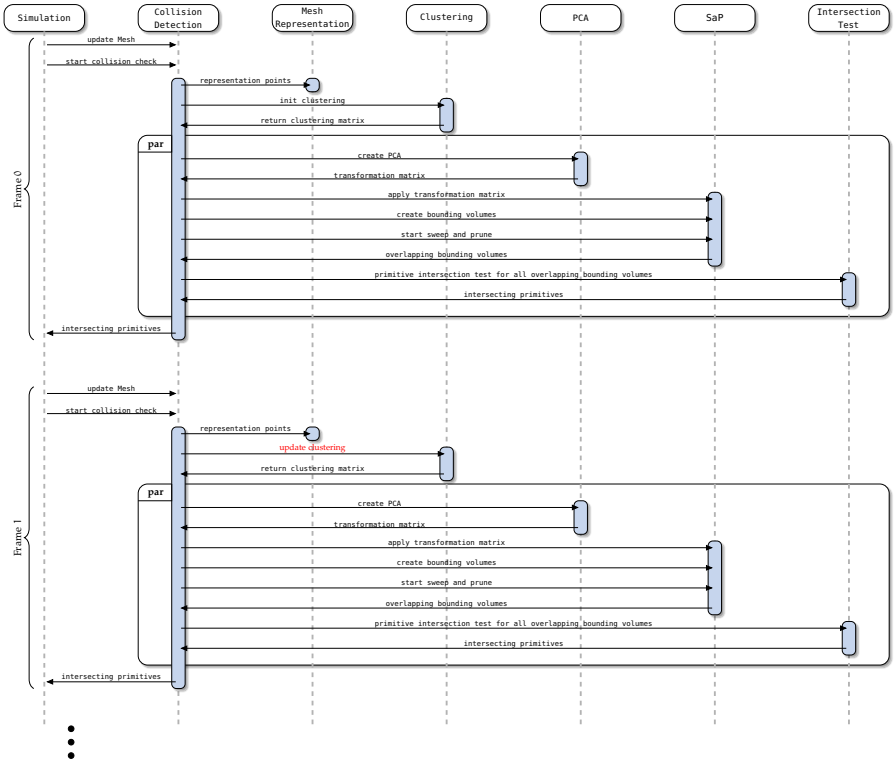


Figure 6.2: Sequence diagram illustrating the most important object interactions arranged in time sequence.

The overall process starts by an update call of the object representation—in our case a 3 D triangle mesh. After the 3 D triangle mesh is sent from the `Simulation` to the `Collision Detection`, the `Simulation` can trigger the process to determine all intersecting primitives. Now our collision detection will perform all steps to make all intersecting primitives available to the `Simulation`. At the beginning our collision detection calls `Mesh Representation` to calculate the data points to represent the 3 D triangle mesh. These data points are used within the `Clustering` process. Every cluster, from the `Clustering` process, can be performed in parallel (see Figure 6.2 the rectangular box labeled with **par**). For each cluster we call the `PCA` and `SaP` process to determine all overlap-

ping BVs. All primitives, which BVs overlap, have to be investigated in more detail. Therefore, we send these primitives to the `Intersection Test` process. Every cluster send all intersecting primitives back to the `Collision Detection`, which collects all primitives from all parallel processes and feed these information back into the `Simulation`. All frames from 2 onwards are using an update function in the `Clustering` process (see Figure 6.2 `update clustering` function colored in red and Section 5.1 for more details) to speedup the collision detection process.

6.3 IMPLEMENTATION

The collision detection approach, including all approaches presented in this work, all applications, and the Graphical User Interface (GUI) are implemented in the programming language C++/CUDA. The main reasons are the high efficiency with respect to computation time and all libraries are available for C++, especially CUDA. We used CUDA because, at the time when we started developing, its functionality outdid by far OpenCL.

Libraries for collision detection process (*required*)	
Boost	lexical cast and tuple data type support
Thrust	parallel sorting, prefix sum and reduction algorithm
CUDA	programming language for the implementation of our massive parallel algorithms

Libraries for visualization only (*not required*)	
Qt	GUI and platform independent file system access
OpenGL	visualization and rendering of the 3 D scene

Table 6.1: A full listing of all used libraries.

6.4 BULLET PHYSICS 2.78

Bullet is an open source real-time physics simulation (under zlib license) developed by Erwin Coumans. Bullet's main features are the support of discrete and continuous collision detection including ray and convex sweep tests. Its supporting concave and convex meshes and all basic primitives for the collision tests. Furthermore, it provides a fast and stable rigid body dynamics constraint solver, vehicle

dynamica, character controller and slider, hinge, generic 6-Degree of Freedom (DoF) and cone twist constraint for ragdolls.

The Bullet Library has been used in many games and movie productions, e. g., the Toy Story 3 game developed for Sony PlayStation 3, Microsoft XBox 360 and Nintendo Wii, use Bullet physics. The digital visual effects company Weta Digital used Bullet's rigid body simulation in the A-Team movie.

6.4.1 *Disadvantages*

Bullet 2.78 supports soft body simulation, e. g., it provides dynamics for cloth, rope and deformable volumes with two-way interaction with rigid bodies. However, there is no support for collision detection between two or more soft bodies. Collision detection between rigid bodies and one soft body object is supported. But most real-world simulations contains more than just one soft body object. Therefore, collision detection between many soft bodies should be taken into account.

In order to overcome this drawback, we integrated our collision detection approach into the Bullet Software Development Kit (SDK). We will describe this implementation in detail in the following section.

6.5 INTEGRATION INTO BULLET PHYSICS

To replace Bullet's own collision detection we have to ensure that our collision detection approach has to be called instead of their built-in. Therefore, we have to replace all function calls of Bullet's collision detection inside `btCollisionWorld::performDiscreteCollisionDetection()` (see Listing 6.1). Thus, Bullet passes all objects within the scene, stored in the array `m_collisionObj`, to our collision detection approach.

```
/** Set our collision detection algorithm as default collision detection
 *  algorithm for all discrete collision checks
 */
void btCollisionWorld::performDiscreteCollisionDetection()
{
  if( ! m_tucIsInit )
    m_tucIsInit = m_tucCollDetHandler.initCollDet( m_collisionObj );
  /* perform collision detection test for all objects */
  m_tucCollDetHandler.doCollisionCheck( m_collisionObj );
}
```

Listing 6.1: Register our collision detection in Bullet (`btCollisionWorld.cpp`)

After each simulation step Bullet perform a collision detection test. Therefore, Bullet calls the algorithm registered in the function `btCollisionWorld::performDiscreteCollisionDetection()`. In the first call our collision detection approach perform an initialization step. In this step we initialize the CUDA device and allocate memory for triangles,

BVs and prefix computation results. Therefore, we use the helper function `void tucBtReplaceColl::initNumberFaces(...)`, which determines the number of triangles per object and verifies the type of each object. At the moment, only two Bullet object types are supported by our implementation, a triangle mesh (`TRIANGLE_MESH_SHAPE_PROXYTYPE`) for rigid body representation and soft bodies (`SOFTBODY_SHAPE_PROXYTYPE`). To perform a collision detection test we have to execute `void tucBtReplaceColl::doCollisionCheck(...)`. Within this function we copy the new vertices—after a simulation step some objects may move or deform—from Bullet to the GPU and perform a collision test for all objects at the same time. Once all of the intersecting primitives of all objects have been determined, we pass these contact points to Bullet. Now Bullet can perform a simulation step and resolve all intersections.

```
/** Replace bullets collision detection
 *    supported bullet object types
 *      - triangle mesh -- TRIANGLE_MESH_SHAPE_PROXYTYPE
 *      - soft bodies   -- SOFTBODY_SHAPE_PROXYTYPE
 */
class tucBtReplaceColl
{
 typedef boost::tuple<uint, uint> collPair;
public:
  ...
  /* initialize our coll det (init cuda, memory allocation, ...) */
  bool initCollDet(
         btAlignedObjectArray<btCollisionObject*>  collisionObjects );
  /* perform collision test and pass contact points to Bullet */
  bool doCollisionCheck(
         btAlignedObjectArray<btCollisionObject*>  collisionObjects );

private:
  /* get vertices from Bullet and copy to GPU */
  void updateVertices(
         btAlignedObjectArray<btCollisionObject*>  collisionObjects );
  /* check objects type and count the number of faces */
  void initNumberFaces(
         btAlignedObjectArray<btCollisionObject*>  collisionObjects );
  ...
```

Listing 6.2: Provide all functions to perform a collision detection check and pass points of contact to Bullet.

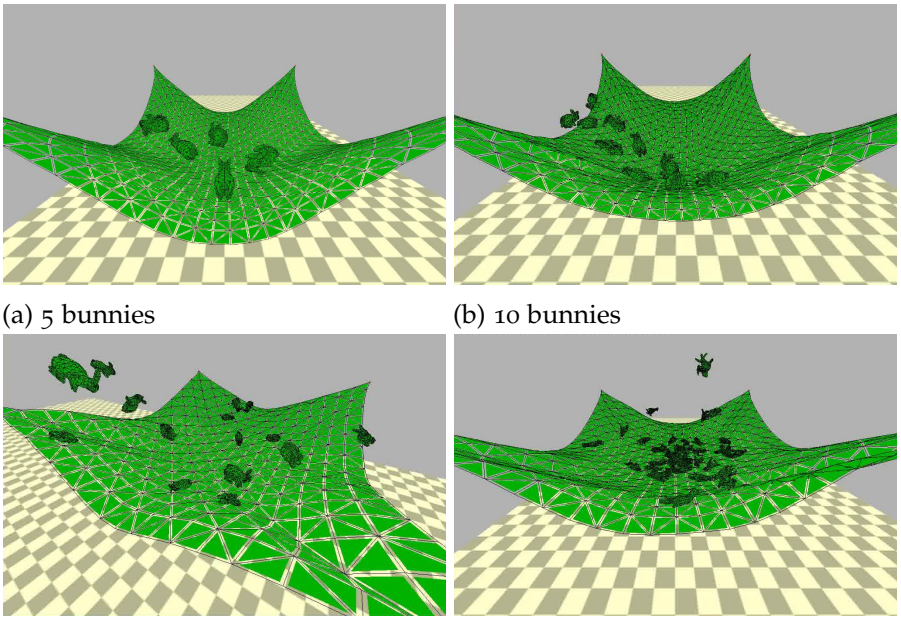
6.5.1 *Disadvantages*

Since Bullet 2.78 is executed entirely on the CPU and our collision detection is executed completely on the GPU, additional communication is necessary. So we have to copy all vertices from the host to the device after each simulation step. Even if we use pinned memory on the host side, memory transfer from host to device is very time

consuming and slow down the whole simulation process. Due to the fact, that Bullet does not run on the GPU we have to perform this memory transfer.

6.6 OUR COLLISION DETECTION IN ACTION

Since we replaced Bullet's built-in collision detection, it is now possible to detect collisions between two soft bodies and between soft and rigid bodies. Figure 6.3 shows some simulations with many soft bodies falling down into a netting. Like you can see all objects within the scene are soft bodies and they generally deform. Therefore, we



(a) 5 bunnies (b) 10 bunnies

(c) 20 bunnies (d) 50 gloves

Figure 6.3: Bullet physics using our collision detection approach for soft body collision detection. Different soft bodies are falling into a netting. Thus the models are strongly deforming which results in a huge amount of inter- and intra-collisions.

get the objects vertices from Bullet and move them on the GPU (see Listing 6.2). As we mentioned in the Section 2.3 we have to determine *all* contact points and intersecting primitives, to handle resting contact and deformations in a physically correct way, and pass these primitives to Bullet. Bullet solves all the collisions and move the primitives depending on their material properties, like stiffness, bending constraints, and more.

7 PERORATION

In this chapter we will recap the main contributions presented in this dissertation and furthermore, we will show up where the development in the field of collision detection can go. Since all chapters have a summary section (see Sections 4.7, 5.1.2, 5.7.5 and 5.8) we will general outline the main contributions and results of our new approach to provide an overall picture. We would like to mention that the *best* collision detection approach and therefore, the focus and the aim behind, is very much dependent upon the underlying problem. It would be true to say that there will be no *universal* collision detection approach for all requirements. The focus of this work is on collision detection in a highly dynamic virtual environment between deformable objects. Furthermore, we want to look ahead and spot and discuss future trends and developments in the field of collision detection and related fields.

7.1 SUMMARY

Interactions with objects and between objects within a virtual environment are not possible without a collision detection system and therefore, collision detection is one of the fundamental technology in interactive virtual simulations. This dissertation introduced a new approach for collision detection of dynamically deforming objects with support for inter-object and intra-object collision detection. Most rigid body collision detection approaches use static acceleration data structures, like a Bounding Volume Hierarchy (BVH) or Inner Sphere Tree (IST). Since static acceleration data structures can become invalid after a deformation process, our approach avoid such static data structures. To ensure that a simulation can compute physically correct forces, self-collisions and resting contact have to be taken into account in the case of soft body simulation (see Section 2.3). To recognize all intersecting primitives our approach works on the primitives directly. In order to avoid unnecessary primitive intersection tests, e. g., primitives which are far apart from each other, we decide to use a topological method, or more precisely a modified Sweep-and-Prune (SaP) approach, to exclude non-colliding primitives as early as possible (see Section 2.7). As sweep direction we decide to use the direction of the first component of the Principal Component Analysis (PCA), because the direction of the first principal component maximizes the variance of primitives, after projection. Furthermore, the direction of the first component of the PCA will change automatically when the object

representation changes after a deformation or moving process. As a consequence we will get a good sweep direction over the whole simulation process. Using the first principal component as sweep direction only, will nevertheless produce false-positives, because of the dimensional reduction in the SaP step (see Section 5.2.1). To eliminate this kind of false-positives we subdivide the scene into connected components using fuzzy c-means algorithm. Our algorithm is especially tailored to be executed massively parallel and therefore, we decide to use the Graphics Processing Unit (GPU) as computing device. Furthermore, the GPUs were still growing exponentially in performance due to massive parallelism, while Central Processing Units (CPUs) hit the serial performance ceiling. Thus, parallelism seems to be a forward-looking technology of increasing performance and therfore, parallelism was one of the most important requirements while developing our new collision detection approach.

In this dissertation, we make several contributions to the area of collision detection, in particular with regard to deformable collision detection, the most important are using PCA to determine a good sweep direction for SaP process and fuzzy c-means for the subdivision process in the context of a highly interactive and deformable virtual environment. These features make it possible to run simulations with deformable objects containing a huge amount of primitives at interactive rates. Furthermore, there are no restrictions relating to the kind of deformation or movement, or conditions, such as for example, the objects have to be watertight. Thanks to the clustering process, each cluster can be distributed to a different device and therefore, theoretical all clusters can be investigated in parallel. The results show that our algorithm is as fast as other state-of-the-art approaches but even more flexible to use more than one GPU only.

7.2 WHERE THE JOURNEY CAN GO?

As stated already in the INTRODUCTION, virtual environments are getting more and more complex and objects become more detailed and accurate. As a result of this, the number of primitives—in the case of a polygonal object representation (see Chapter 2)—has increase significantly in the last years. Furthermore, new virtual technologies will become part of our daily life, like the Oculus Rift or Google Glass. The Oculus Rift, a virtual reality head-mounted display, opens up a wide range of people the possibility to plunge into a virtual environment. To simulate a fully immersive and interactive virtual environment, where the user cannot differentiate between the real-world and the virtual one, objects have to interact in the virtual world in the same way like they interact in the real-world. Therefore, a physically correct behavior of the objects is necessary to guarantee a high degree of immersion. Currently a major problem is that in

the real-world nearly all objects can deform or will break up in its individual components, if the forces applied to the objects are strong enough. Thus, soft bodies containing a huge amount of primitives are becoming increasingly important and new collision detection approaches should support this type of object.

7.2.1 *Quality of Contact Information*

Another important objective of the collision handling process is to compute accurate contact information (penetration depth, penetration volume, or other) for a physically correct or plausible collision response. To this day there is no approach for deformable objects which can compute the penetration volume in real-time but according to Fisher and Lin [FL01, Section 5.1], this penetration measure is "the most complicated yet accurate method" to define the extent of intersection. Thus, an approach which determines the penetration volume for deformable objects in a very fast way will improve the quality of the computed forces and therefore, the quality of the physical simulation.

7.2.2 *Point Clouds*

As aforementioned, point clouds become more and more important because 3 D scanners are seen in everyday life, e. g., Microsoft's Kinect, Google's Project Tango or a simple webcam [Iza+11; PRD09; RHL02]. Therefore, collision detection approaches for point clouds will come into the focus in the future. Since most 3 D scanners provide point cloud data in real-time, static data structures are not suitable in most cases because they need an update step in every frame. This problem is closely related to soft body collision detection and can profit from the insights gained. The first step is a surface triangulation on point clouds with normals, to obtain a triangle mesh based on projections of the local neighborhoods [MRB09]. This triangle mesh can now be handle by a fast soft body collision detection or a rigid body collision detection approach which provides an update step for the underlaying acceleration data structure. But till today the triangulation process cannot be performed in real-time for huge point clouds.

7.2.3 *Haptics*

Besides rendering of visual and aural information of the virtual environment, force-feedback is another important technique to significantly increase the degree of immersion and usability within a virtual environment. Botden, Torab, Buzink, and Jakimowicz [Bot+08] and Våpenstad, Hofstad, Langø, Mårvik, and Chmarra [Våp+13] showed

that haptic sensations influence psychomotor performance in virtual tasks like laparoscopy. Furthermore, the kind of force-feedback device is important for virtual interactions. Weller and Zachmann [WZ12] showed that 6-DoF force-feedback devices outperform 3-DoF devices significantly, both in user perception and in user performance. However, nowadays haptic devices are still unhandy, very expensive and very hard to installation and handling.

Since human being is very sensitive to the haptic feedback—the temporal resolution of the human tactile sense is very high—a constant update rate of 1000 Hz is needed for hard surfaces will feel 100 % realistic [Mar+96]. Thus, the collision detection process has to be very fast to reach this update rate. Until now, no exact collision detection approach for deformable objects consists of tens of thousands of primitives is fast enough to provide a collision response containing forces and torques at this high frequency.

7.2.4 *Natural Interaction*

Most current simulations use much simplified models, e. g., to represent the human body or especially the human hand. Microsoft's Kinect is able to track the whole body with fairly well accuracy. The next consequent step is the precise tracking. Mohr [Moh12] presented techniques to detect and track the full-DoF human hand motion using conventional cameras. Furthermore, they presented an artificial hand model and an approach for a very compact hand motion description. We are entirely certain that future developments will allow a precise tracking of the whole human body, including hand, fingers, legs, and eyes.

There is currently no physical correct or plausible simulation model available, that allows complex interactions between the human model and objects, e. g., pick up things from the floor with the index finger and the thumb. In most current applications objects are pinned to the virtual hand, if the objects are located in the immediate vicinity of the virtual hand. In order to realize such precise interactions two factors are important: a detailed physically-based deformable hand model, and an accurate simulation of the fingers' frictional forces. To determine these forces an exact collision detection approach for deformable models, which takes all kind of contacts into account, e. g., collision, self-collision, resting or persistent contacts, is needed.

APPENDIX

A

REFERENCE SHEETS

TYPE	CUDA TERMINOLOGY	OPENCL TERMINOLOGY
Program and Hardware	GPU	Device
	Multiprocessor	Compute Unit
	Scalar Core	Processing Element
	Kernel	Kernel
	Host Program	Host Program
	Grid	NDRange
	Thread-Block	Work-Group
	Thread	Work-Item
Memory	Global Memory	Global Memory
	Constant Memory	Constant Memory
	Texture Memory	Texture Memory
	Shared (per-block) Memory	Local Memory
	Local Memory	Private Memory

Table A.1: Terminology overview used by NVIDIA CUDA and OpenCL developed by Khronos Group [KW10; KP12].

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

[Kös+09] Michael Köster, Peter Novák, David Mainzer, and Bernd Fuhrmann. "Two Case Studies for Jazzyk BSM". In: *Proceedings of Agents for Games and Simulations, AGS 2009, AAMAS 2009 co-located workshop*. Volume 5920. LNAI. Springer Verlag, 2009, pages 31–45. URL: http://link.springer.com/book/10.1007/978-3-642-11198-3.

[MWZ11] David Mainzer, René Weller, and Gabriel Zachmann. "Kollisionserkennung und natürliche Interaktion in vir"-tu"-ellen Umgebungen". In: *Virtuelle Techniken im industriellen Umfeld*. Edited by Werner Schreiber und Peter Zimmermann. Springer, 2011. Chapter 3.2, 3.4, pages 33–38, 114–116. ISBN: 978-3-642-20635-1. URL: http://www.springer.com/engineering/signals/book/978-3-642-20635-1.

[MZ13] David Mainzer and Gabriel Zachmann. "CDFC: Collision Detection Based on Fuzzy Clustering for Deformable Objects on GPUs". In: *WSCG 2013 - POSTER Proceedings*. Volume 21. 3. Poster. Plzeň, Czech Republic, July 2013, pages 5–8.

[MZ14] David Mainzer and Gabriel Zachmann. "Collision Detection Based on Fuzzy Scene Subdivision". In: *Symposium on GPU Computing and Applications (Singapore, 2013)*. Edited by Yiyu Cai and Simon See. Volume 3. Springer, 2014. URL: https://www.springer.com/engineering/signals/book/978-981-287-133-6.

[Wel+10] René Weller, David Mainzer, Mikel Sagardia, Thomas Hulin, Gabriel Zachmann, and Carsten Preusche. "A benchmarking suite for 6-DOF real time collision response algorithms". In: *Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology (VRST)*. Hong Kong: ACM, Nov. 2010, pages 63–70. ISBN: 978-1-4503-0441-2. DOI: http://doi.acm.org/10.1145/1889863.1889874. URL: http://cg.in.tu-clausthal.de/publications.shtml%5C#vrst2010 (cited on page 89).

[Wel+14] René Weller, David Mainzer, Abhishek Srinivas, Matthias Teschner, and Gabriel Zachmann. "Massively Parallel Batch Neural Gas for Bounding Volume Hierarchy Construction". In: *Virtual Reality Interactions and Physical Simulations (VRIPhys)*. Bremen, Germany: Eurographics Association, Sept. 2014 (cited on pages 38, 58).

BIBLIOGRAPHY

[AEG98] Pankaj K. Agarwal, Jeff Erickson, and Leonidas J. Guibas. "Kinetic Binary Space Partitions for Intersecting Segments and Disjoint Triangles". In: *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '98. San Francisco, California, USA: Society for Industrial and Applied Mathematics, 1998, pages 107–116. ISBN: 0-89871-410-9. URL: http://dl.acm.org/citation.cfm?id=314613.314688 (cited on page 45).

[Aga+04] Pankaj Agarwal, Leonidas Guibas, An Nguyen, Daniel Russel, and Li Zhang. "Collision Detection for Deforming Necklaces". In: *Computational Geometry: Theory and Applications* 28 (2004), pages 137–163 (cited on page 57).

[Amd67] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pages 483–485. DOI: 10.1145/1465482.1465560. URL: http://doi.acm.org/10.1145/1465482.1465560 (cited on pages 3, 40).

[And73] Michael R. Anderberg. *Cluster Analysis for Applications*. Technical report. DTIC Document, 1973 (cited on page 48).

[AT11] Nathan Andrysco and Xavier Tricoche. "Implicit and Dynamic Trees for High Performance Rendering". In: *Proceedings of Graphics Interface 2011*. GI '11. St. John's, Newfoundland, Canada: Canadian Human-Computer Communications Society, 2011, pages 143–150. ISBN: 978-1-4503-0693-5. URL: http://dl.acm.org/citation.cfm?id=1992917.1992941 (cited on page 46).

[AMoo] Ulf Assarsson and Tomas Möller. "Optimized View Frustum Culling Algorithms for Bounding Boxes". In: *Journal of Graphics Tools* 5.1 (2000), pages 9–22 (cited on page 33).

[AGA10] Quentin Avril, Valérie Gouranton, and Bruno Arnaldi. "A Broad Phase Collision Detection Algorithm Adapted to Multi-cores Architectures". In: *VRIC 2010 Proceedings* 12 (2010) (cited on page 20).

[A+09] Quentin Avril, Valérie Gouranton, Bruno Arnaldi, et al. "New Trends in Collision Detection Performance". In: *VRIC'09 Proceedings* 11 (2009) (cited on page 14).

[BWG03] Kavita Bala, Bruce Walter, and Donald P. Greenberg. "Combining Edges and Points for Interactive High-Quality Rendering". In: *ACM Transactions on Graphics (TOG)*. Volume 22. 3. ACM. 2003, pages 631–640 (cited on page 11).

[Bar92] David Baraff. "Dynamic Simulation of Non-Penetrating Rigid Bodies". PhD thesis. Cornell University, 1992 (cited on pages 20, 28).

[BW92] David Baraff and Andrew Witkin. *Dynamic Simulation of Non-penetrating Flexible Bodies*. Volume 26. 2. ACM, 1992 (cited on page 14).

[BWK03] David Baraff, Andrew Witkin, and Michael Kass. "Untangling Cloth". In: *ACM Transactions on Graphics (TOG)*. Volume 22. 3. ACM. 2003, pages 862–870 (cited on page 4).

[Bar+96] Gill Barequet, Bernard Chazelle, Leonidas J. Guibas, Joseph S. B. Mitchell, and Ayellet Tal. "BOXTREE: A Hierarchical Representation for Surfaces in 3D". In: *Computer Graphics Forum*. Volume 15. 3. Wiley Online Library. 1996, pages 387–396 (cited on page 24).

[Bar97] Anthony C. Barkans. "High Quality Rendering Using the Talisman Architecture". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. HWWS '97. Los Angeles, California, USA: ACM, 1997, pages 79–88. ISBN: 0-89791-961-0. DOI: 10.1145/258694.258722. URL: http://doi.acm.org/10.1145/258694.258722 (cited on page 8).

[BHW96] Ronen Barzel, John F. Hughes, and Daniel N. Wood. "Plausible Motion Simulation for Computer Graphics Animation". In: *Proceedings of the Eurographics Workshop on Computer Animation and Simulation '96*. Poitiers, France: Springer-Verlag New York, Inc., 1996, pages 183–197. ISBN: 3-211-82885-0. URL: http://dl.acm.org/citation.cfm?id=274976.274989 (cited on page 33).

[Bec+90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. *The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles*. Volume 19. 2. ACM, 1990 (cited on pages 24, 27).

[BS09] Norbert Beckmann and Bernhard Seeger. "A Revised R*-tree in Comparison with Related Index Structures". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM. 2009, pages 799–812 (cited on page 27).

[BH11] Nathan Bell and Jared Hoberock. "Thrust: A Productivity-Oriented Library for CUDA". In: *GPU Computing Gems* 7 (2011) (cited on pages 58, 59, 78).

[Ben75] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching". In: *Communications of the ACM* 18.9 (1975), pages 509–517 (cited on pages 19, 46).

[BF79] Jon Louis Bentley and Jerome H. Friedman. "Data Structures for Range Searching". In: *ACM Computing Surveys (CSUR)* 11.4 (1979), pages 397–409 (cited on page 19).

[Ber+08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. ISBN: 3540779736, 9783540779735 (cited on pages 28, 33, 47).

[Ber98] Gino van den Bergen. "Efficient Collision Detection of Complex Deformable Models Using AABB Trees". In: *J. Graph. Tools* 2.4 (Jan. 1998), pages 1–13. ISSN: 1086-7651. DOI: 10.1080/10867651.1997.10487480. URL: http://dx.doi.org/10.1080/10867651.1997.10487480 (cited on pages 23, 25).

[Ber04] Gino van den Bergen. "Ray Casting against General Convex Objects with Application to Continuous Collision Detection". In: *Submitted to Journal of Graphics Tools* (2004) (cited on page 30).

[Bez81] James C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms*. Norwell, MA, USA: Kluwer Academic Publishers, 1981. ISBN: 0306406713 (cited on pages 48, 50, 75).

[Bot+08] Sanne MBI Botden, Fawaz Torab, Sonja N Buzink, and Jack J Jakimowicz. "The importance of haptic feedback in laparoscopic suturing training and the additive value of virtual reality simulation". In: *Surgical endoscopy* 22.5 (2008), pages 1214–1222 (cited on page 115).

[BK04] Mario Botsch and Leif Kobbelt. "An Intuitive Framework for Real-Time Freeform Modeling". In: *ACM Transactions on Graphics (TOG)*. Volume 23. 3. ACM. 2004, pages 630–634 (cited on page 17).

[BK05] Mario Botsch and Leif Kobbelt. "Real-Time Shape Editing using Radial Basis Functions". In: *Computer graphics forum*. Volume 24. 3. Wiley Online Library. 2005, pages 611–621 (cited on page 17).

[BB95] Leon Bottou and Yoshua Bengio. "Convergence Properties of the K-Means Algorithms". In: *Advances in Neural Information Processing Systems 7*. 1995 (cited on page 50).

[BO04] Gareth Bradshaw and Carol O'Sullivan. "Adaptive Medial-axis Approximation for Sphere-tree Construction". In: *ACM Transactions on Graphics* 23.1 (Jan. 2004), pages 1–26. ISSN: 0730-0301. DOI: 10.1145/966131.966132. URL: http://doi.acm.org/10.1145/966131.966132 (cited on page 55).

[Bra08] Peter Brass. *Advanced Data Structures*. 1st edition. New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521880378, 9780521880374 (cited on page 47).

[BFA02] Robert Bridson, Ronald Fedkiw, and John Anderson. "Robust Treatment of Collisions, Contact and Friction for Cloth Animation". In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '02. San Antonio, Texas: ACM, 2002, pages 594–603. ISBN: 1-58113-521-1. DOI: 10.1145/566570.566623. URL: http://doi.acm.org/10.1145/566570.566623 (cited on pages 4, 15, 16).

[Bro+10] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. "State-of-the-art in Heterogeneous Computing". In: *Sci. Program.* 18.1 (Jan. 2010), pages 1–33. ISSN: 1058-9244. URL: http://dl.acm.org/citation.cfm?id=1804799.1804800 (cited on page 3).

[BHS13] André R. Brodtkorb, Trond R. Hagen, and Martin L. SæTra. "Graphics Processing Unit (GPU) Programming Strategies and Trends in GPU Computing". In: *J. Parallel Distrib. Comput.* 73.1 (Jan. 2013), pages 4–13. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2012.04.003. URL: http://dx.doi.org/10.1016/j.jpdc.2012.04.003 (cited on page 3).

[Buc+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. "Brook for GPUs: Stream Computing on Graphics Hardware". In: *ACM Transactions on Graphics (TOG)*. Volume 23. 3. ACM. 2004, pages 777–786 (cited on page 34).

[Can86] John Canny. "Collision Detection for Moving Polyhedra". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 2 (1986), pages 200–209 (cited on page 30).

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. "The Ray Engine". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '02.

Saarbrucken, Germany: Eurographics Association, 2002, pages 37–46. ISBN: 1-58113-580-7. URL: http://dl.acm.org/citation.cfm?id=569046.569052 (cited on pages 34, 38).

[Cha84] Bernard Chazelle. "Convex Partitions of Polyhedra: A Lower Bound and Worst-Case Optimal Algorithm". In: *SIAM Journal on Computing* 13.3 (1984), pages 488–507 (cited on page 12).

[CWK03] Yi-King Choi, Wenping Wang, and Myung-Soo Kim. "Exact Collision Detection of Two Moving Ellipsoids under Rational Motions". In: *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*. Volume 1. IEEE. 2003, pages 349–354 (cited on page 23).

[CS92] Yiorgos Chrysanthou and Mel Slater. "Computing Dynamic Changes to BSP Trees". In: *Computer Graphics Forum*. Volume 11. 3. Wiley Online Library. 1992, pages 321–332 (cited on page 20).

[CW96] Kelvin Chung and Wenping Wang. "Quick Collision Detection of Polytopes in Virtual Environments". In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST 96)*. 1996, pages 125–131 (cited on page 26).

[Cla76] James H. Clark. "Hierarchical Geometric Models for Visible Surface Algorithms". In: *Communications of the ACM* 19.10 (1976), pages 547–554 (cited on page 33).

[Cla94] Kenneth L. Clarkson. "More Output-Sensitive Geometric Algorithms". In: *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*. IEEE. 1994, pages 695–702 (cited on page 8).

[Coh+95] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi. "I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments". In: *Proceedings of the 1995 symposium on Interactive 3D graphics*. I3D '95. Monterey, California, United States: ACM, 1995, 189–ff. ISBN: 0-89791-736-7. DOI: 10.1145/199404.199437. URL: http://doi.acm.org/10.1145/199404.199437 (cited on page 20).

[Con43] Luigi Frederico Conte di Menabrea. "Sketch of the Analytical Engine Invented by Charles Babbage from the Bibliothèque Universelle de Genève, October, 1842, No. 82". In: *Scientific Memoirs, (Translated with notes by Ada Augusta Lovelace)* (1843) (cited on page 2).

[Coo67] Steven A. Coons. *Surfaces for Computer Aided Design of Space Forms*. Technical report. DTIC Document, 1967 (cited on page 11).

[Cot+06] Marie Cottrell, Barbara Hammer, Alexander Hasenfuß, and Thomas Villmann. "Batch and Median Neural Gas". In: *Neural Networks* 19 (July 2006), pages 762–771. ISSN: 0893-6080 (cited on pages 53, 54).

[Cou05] Erwin Coumans. *Continuous Collision Detection and Physics*. Technical report. Sony Computer Entertainment, Aug. 2005 (cited on pages 29, 30).

[Cou12] Erwin Coumans. *Bullet Physics Library*. http://bulletphysics.com. 2012 (cited on page 9).

[Cou01] Murilo G. Coutinho. *Dynamic Simulations of Multibody Systems*. Springer, 2001 (cited on page 14).

[Cre10] Ryan Henson Creighton. *Unity 3D Game Development by Example: A Seat-of-Your-Pants Manual for Building Fun, Groovy Little Games Quickly*. Packt Publishing Ltd, 2010 (cited on page 9).

[CTM08] Sean Curtis, Rasmus Tamstorf, and Dinesh Manocha. "Fast Collision Detection for Deformable Models Using Representative-triangles". In: *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*. I3D '08. Redwood City, California: ACM, 2008, pages 61–69. ISBN: 978-1-59593-983-8. DOI: 10.1145/1342250.1342260. URL: http://doi.acm.org/10.1145/1342250.1342260 (cited on page 10).

[Deb+01] Gilles Debunne, Mathieu Desbrun, Marie-Paule Cani, and Alan H. Barr. "Dynamic Real-Time Deformations using Space & Time Adaptive Sampling". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM. 2001, pages 31–36 (cited on page 18).

[DB13] Crispin Deul and Jan Bender. "Physically-Based Character Skinning." In: *VRIPHYS* 13 (2013), pages 25–34 (cited on pages 18, 78).

[Dje+07] Peter Djeu, Warren Hunt, Rui Wang, Ikrima Elhassan, Gordon Stoll, and William R. Mark. *Razor: An Architecture for Dynamic Multiresolution Ray Tracing*. Computer Science Department, University of Texas at Austin, 2007 (cited on page 34).

[Don+98] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. Van der Vorst. *Numerical linear algebra on high-performance computers*. Volume 7. Siam, 1998 (cited on page 42).

[Dru+10] Evan Drumwright, John Hsu, Nathan Koenig, and Dylan Shell. "Extending Open Dynamics Engine for Robotics Simulation". In: *Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2010, pages 38–50 (cited on page 9).

[Dub87] Richard C. Dubes. "How Many Clusters Are Best?&Mdash;an Experiment". In: *Pattern Recogn.* 20.6 (Nov. 1987), pages 645–663. ISSN: 0031-3203. DOI: 10.1016/0031-3203(87)90034-3. URL: http://dx.doi.org/10.1016/0031-3203(87)90034-3 (cited on page 48).

[DHS12] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. John Wiley & Sons, 2012 (cited on pages 48, 50).

[Ebe00] David Eberly. "Intersection of Cylinders". In: *Geometric Tools, Inc* (2000) (cited on page 23).

[Ebe02] David Eberly. "Intersection of a Sphere and a Cone". In: *Geometric Tools. Inc* (2002) (cited on page 23).

[Ebe03] David H. Eberly. *Game Physics*. Elsevier, 2003 (cited on page 14).

[ES99] Jens Eckstein and Elmar Schömer. "Dynamic Collision Detection in Virtual Reality Applications". In: *Proc. The 7-th Int'l Conf. in Central Europe on Comp. Graphics, Vis. and Interactive Digital Media'99 (WSCG'99)*. Citeseer. 1999, pages 71–78 (cited on pages 7, 30).

[EM81] Herbert Edelsbrunner and Hermann A. Maurer. "On the Intersection of Orthogonal Objects". In: *Information Processing Letters* 13.4 (1981), pages 177–181 (cited on page 19).

[Eri05] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series In Interactive 3-D Technology)*. 1st edition. City: Morgan Kaufmann, Jan. 2005, page 632. ISBN: 1558607323 (cited on pages 2, 5, 7, 8, 13, 14, 17, 19, 74).

[Far+08] Reza Farivar, Daniel Rebolledo, Ellick Chan, and Roy H. Campbell. "A Parallel Implementation of K-Means Clustering on GPUs." In: *PDPTA*. 2008, pages 340–345 (cited on page 52).

[Fau+12] François Faure, Christian Duriez, Hervé Delingette, Jérémie Allard, Benjamin Gilles, Stéphanie Marchesseau, Hugo Talbot, Hadrien Courtecuisse, Guillaume Bousquet, Igor Peterlik, et al. "SOFA: A Multi-Model Framework for Interactive Physical Simulation". In: *Soft Tissue Biomechanical Modeling for Computer Assisted Surgery*. Springer, 2012, pages 283–321 (cited on page 9).

[Fig+10] Mauro Figueiredo, João Oliveira, Bruno Araújo, and João Pereira. "An Efficient Collision Detection Algorithm for Point Cloud Models". In: *20th International conference on Computer Graphics and Vision*. Volume 43. Citeseer. 2010, page 44 (cited on page 12).

[FL01] Susan Fisher and Ming C. Lin. "Fast Penetration Depth Estimation for Elastic Bodies Using Deformed Distance Fields". In: *Proc. International Conf. on Intelligent Robots and Systems (IROS)*. 2001, pages 330–336 (cited on pages 54, 115).

[LMT91] Benoit La-fleur, Nadia Magnenat-Thalmann, and Daniel Thalmann. "Cloth Animation with Self-Collision Detection". In: *Modeling in Computer Graphics*. Springer, 1991, pages 179–187 (cited on page 15).

[Fly72] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Trans. Comput.* 21.9 (Sept. 1972), pages 948–960. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071. URL: http://dx.doi.org/10.1109/TC.1972.5009071 (cited on page 3).

[Fol+94] James D. Foley, Andries Van Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. *Introduction to Computer Graphics*. Volume 55. Addison-Wesley Reading, 1994 (cited on pages 8, 14).

[FS05] Tim Foley and Jeremy Sugerman. "KD-tree Acceleration Structures for a GPU Raytracer". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '05. Los Angeles, California: ACM, 2005, pages 15–22. ISBN: 1-59593-086-8. DOI: 10.1145/1071866.1071869. URL: http://doi.acm.org/10.1145/1071866.1071869 (cited on page 34).

[FAW10] R. Fraedrich, S. Auer, and R. Westermann. "Efficient High-Quality Volume Rendering of SPH Data". In: *Visualization and Computer Graphics, IEEE Transactions on* 16.6 (Nov. 2010), pages 1533–1540. ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.148 (cited on page 36).

[FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. "On Visible Surface Generation by a Priori Tree Structures". In: *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '80. Seattle, Washington, USA: ACM, 1980, pages 124–133. ISBN: 0-89791-021-4. DOI: 10.1145/800250.807481. URL: http://doi.acm.org/10.1145/800250.807481 (cited on page 19).

[Fuk90] Keinosuke Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic press, 1990 (cited on page 48).

[GDO00] Fabio Ganovelli, John Dingliana, and Carol O'Sullivan. "BucketTree: Improving Collision Detection Between Deformable Objects". In: *Proc. of Spring Conference on Computer Graphics SCCG'oo*. Volume 11. 2000 (cited on page 19).

[Gär99] Bernd Gärtner. "Fast and Robust Smallest Enclosing Balls". In: *ESA*. Edited by Jaroslav Nesetril. Volume 1643. Lecture Notes in Computer Science. Springer, 1999, pages 325–338. ISBN: 3-540-66251-0. URL: http://link.springer.de/link/service/series/0558/bibs/1643/16430325.htm (cited on page 57).

[Gay+05] Russell Gayle, Paul Segars, Ming C. Lin, and Dinesh Manocha. "Path Planning for Deformable Robots in Complex Environments". In: *Robotics: science and systems*. Volume 2005. Citeseer. 2005, pages 225–232 (cited on page 33).

[GF90] Elmer G. Gilbert and C.-P. Foo. "Computing the Distance Between General Convex Objects in Three-Dimensional Space". In: *Robotics and Automation, IEEE Transactions on* 6.1 (1990), pages 53–61 (cited on page 26).

[GJK88] Elmer G. Gilbert, Daniel W. Johnson, and S. Sathiya Keerthi. "A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space". In: *Robotics and Automation, IEEE Journal of* 4.2 (1988), pages 193–203 (cited on page 26).

[GS87] Jeffrey Goldsmith and John Salmon. "Automatic Creation of Object Hierarchies for Ray Tracing". In: *Computer Graphics and Applications, IEEE* 7.5 (1987), pages 14–20 (cited on pages 24, 34).

[GV12] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Volume 3. JHU Press, 2012 (cited on page 74).

[GH77] A. D. Gordon and J. T. Henderson. "An Algorithm for Euclidean Sum of Squares Classification". In: *Biometrics* (1977), pages 355–362 (cited on page 50).

[GSF99] Craig Gotsman, Oded Sudarsky, and Jeffrey A. Fayman. "Optimized Occlusion Culling using Five-Dimensional Subdivision". In: *Computers & Graphics* 23.5 (1999), pages 645–654. ISSN: 0097-8493. DOI: http://dx.doi.org/10.1016/S0097-8493(99)00088-6. URL: http://www.sciencedirect.com/science/article/pii/S0097849399000886 (cited on page 33).

[GLM96] Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. "OBBTree: A Hierarchical Structure for Rapid Interference Detection". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pages 171–180. ISBN: 0-89791-746-4. DOI: 10.1145/237170.237244. URL: http://doi.acm.org/10.1145/237170.237244 (cited on pages 10, 23, 24, 92).

[GLM05] Naga K. Govindaraju, Ming C. Lin, and Dinesh Manocha. "Quick-cullide: Fast inter-and intra-object collision culling using graphics hardware". In: *Virtual Reality, 2005. Proceedings. VR 2005. IEEE*. IEEE. 2005, pages 59–66 (cited on page 32).

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. "Hierarchical Z-buffer Visibility". In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM. 1993, pages 231–238 (cited on page 33).

[GGK06] Alexander Greß, Michael Guthe, and Reinhard Klein. "GPU-based Collision Detection for Deformable Parameterized Surfaces". In: *Computer Graphics Forum*. Volume 25. 3. Wiley Online Library. 2006, pages 497–506 (cited on pages 10, 11, 32).

[GZ06] Alexander Greß and Gabriel Zachmann. "GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures". In: *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Rhodes Island, Greece, 25–29 April 2006 (cited on page 38).

[GNZ03] Leonidas J. Guibas, An Nguyen, and Li Zhang. "Zonotopes As Bounding Volumes". In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '03. Baltimore, Maryland: Society for Industrial and Applied Mathematics, 2003, pages 803–812. ISBN: 0-89871-538-5. URL: http://dl.acm.org/citation.cfm?id=644108.644241 (cited on page 23).

[Gus88] John L. Gustafson. "Reevaluating Amdahl's Law". In: *Communications of the ACM* 31 (1988), pages 532–533 (cited on page 3).

[Gut84] Antonin Guttman. *R-Trees: A Dynamic Index Structure for Spatial Searching*. Volume 14. 2. ACM, 1984 (cited on page 19).

[Hah88] James K. Hahn. "Realistic Animation of Rigid Bodies". In: *ACM SIGGRAPH Computer Graphics*. Volume 22. 4. ACM. 1988, pages 299–308 (cited on pages 2, 23).

[HHV06] Barbara Hammer, Alexander Hasenfuss, and Thomas Villmann. "Magnification Control for Batch Neural Gas". In: *ESANN*. 2006, pages 7–12. URL: http://www.dice.ucl.ac.be/Proceedings/esann/esannpdf/es2006-83.pdf (cited on page 56).

[Har05] Mark Harris. "Fast Fluid Dynamics Simulation on the GPU". In: *ACM SIGGRAPH 2005 Courses*. SIGGRAPH '05. Los Angeles, California: ACM, 2005. DOI: 10.1145/1198555.1198790. URL: http://doi.acm.org/10.1145/1198555.1198790 (cited on page 38).

[Har+03] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. "Simulation of Cloud Dynamics on Graphics Hardware". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association. 2003, pages 92–101 (cited on page 38).

[HFB12] A. H. Hassan, Christopher J. Fluke, and David G. Barnes. "A Distributed GPU-based Framework for real-time 3D Volume Rendering of Large Astronomical Data Cubes". In: *Publications of the Astronomical Society of Australia* 29.03 (2012), pages 340–351 (cited on page 36).

[HS89] Trevor Hastie and Werner Stuetzle. "Principal Curves". In: *Journal of the American Statistical Association* 84.406 (1989), pages 502–516 (cited on page 75).

[Hat84] IItevor Hattie. "Principal Curves and Surfaces". In: (1984) (cited on page 75).

[He99] Taosong He. "Fast Collision Detection Using QuOSPO Trees". In: *Proceedings of the 1999 Symposium on Interactive 3D Graphics*. I3D '99. Atlanta, Georgia, USA: ACM, 1999, pages 55–62. ISBN: 1-58113-082-1. DOI: 10.1145/300523.300529. URL: http://doi.acm.org/10.1145/300523.300529 (cited on page 23).

[Hel97] Martin Held. "Erit: A Collection of Efficient and Reliable Intersection Tests". In: *Journal of Graphics Tools* 2.4 (1997), pages 25–44 (cited on pages 23, 81).

[HKM95] Martin Held, James T. Klosowski, and Joseph S. B. Mitchell. "Evaluation of Collision Detection Methods for Virtual Reality Fly-Throughs". In: *In Canadian Conference on Computational Geometry*. 1995, pages 205–210 (cited on pages 7, 19).

[HKM96] Martin Held, James T. Klosowski, and Joseph S. B. Mitchell. "Collision Detection for Fly-throughs in Virtual Environments". In: *Proceedings of the Twelfth Annual Symposium on Computational Geometry*. SCG '96. Philadelphia,

Pennsylvania, USA: ACM, 1996, pages 513–514. ISBN: 0-89791-804-5. DOI: 10.1145/237218.237428. URL: http://doi.acm.org/10.1145/237218.237428 (cited on page 29).

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2012 (cited on page 2).

[H+09] Everton Hermann, Bruno Raffin, François Faure, et al. "Interactive Physical Simulation on Multicore Architectures". In: *Eurographics Workhop on Parallel Graphics and Visualization*. 2009 (cited on page 31).

[Hor+07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. "Interactive k-D Tree GPU Raytracing". In: *Proceedings of the 2007 symposium on Interactive 3D graphics and games*. ACM. 2007, pages 167–174 (cited on page 34).

[HZQ01] Gao Shuming Wan Huagen, Fan Zhaowei, and Peng Qunsheng. "A Parallel Collision Detection Algorithm Based on Hybrid Bounding Volume Hierarchy". In: *CAD/Graphics2001, August* (2001) (cited on page 31).

[Hub95] Philip M. Hubbard. "Collision Detection for Interactive Graphics Applications". In: *IEEE Transactions on Visualization and Computer Graphics* 1.3 (Sept. 1995), pages 218–230 (cited on page 55).

[Hub96] Philip M. Hubbard. "Approximating Polyhedra with Spheres for Time-critical Collision Detection". In: *ACM Trans. Graph.* 15.3 (July 1996), pages 179–210. ISSN: 0730-0301. DOI: 10.1145/231731.231732. URL: http://doi.acm.org/10.1145/231731.231732 (cited on pages 23, 33).

[IMH05] Takeo Igarashi, Tomer Moscovich, and John F. Hughes. "As-rigid-as-possible Shape Manipulation". In: *ACM SIGGRAPH 2005 Papers*. SIGGRAPH '05. Los Angeles, California: ACM, 2005, pages 1134–1141. DOI: 10.1145/1186822.1073323. URL: http://doi.acm.org/10.1145/1186822.1073323 (cited on page 17).

[Int12] Epic Games International. *Unreal Engine Documentation*. https://docs.unrealengine.com/latest/INT/. 2012 (cited on page 9).

[Iza+11] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, and Andrew Fitzgibbon. "KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera". In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*.

UIST '11. Santa Barbara, California, USA: ACM, 2011, pages 559–568. ISBN: 978-1-4503-0716-1. DOI: 10.1145/2047196.2047270. URL: http://doi.acm.org/10.1145/2047196.2047270 (cited on pages 11, 115).

[JMF99] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. "Data Clustering: A Review". In: *ACM computing surveys (CSUR)* 31.3 (1999), pages 264–323 (cited on page 48).

[JP99] Doug L. James and Dinesh K. Pai. "ArtDefo: Accurate Real Time Deformable Objects". In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pages 65–72. ISBN: 0-201-48560-5. DOI: 10.1145/311535.311542. URL: http://dx.doi.org/10.1145/311535.311542 (cited on page 17).

[JT05] Doug L. James and Christopher D. Twigg. "Skinning Mesh Animations". In: *ACM SIGGRAPH 2005 Papers*. SIGGRAPH '05. Los Angeles, California: ACM, 2005, pages 399–407. DOI: 10.1145/1186822.1073206. URL: http://doi.acm.org/10.1145/1186822.1073206 (cited on page 18).

[Jam10] Ondrej Jamriška. "Interactive Ray Tracing of Distance Fields". In: *14th Central European Seminar on Computer Graphics*. Citeseer. 2010, page 91 (cited on page 36).

[JTT01] Pablo Jiménez, Federico Thomas, and Carme Torras. "3D Collision Detection: A Survey". In: *Computers & Graphics* 25.2 (2001), pages 269–285 (cited on page 14).

[Jol05] I. Jolliffe. *Principal Component Analysis*. Wiley Online Library, 2005 (cited on page 74).

[Kan+13] Yoon-Sig Kang, Jae-Ho Nah, Woo-Chan Park, and Sung-Bong Yang. "gkDtree: A Group-based Parallel Update Kd-tree for Interactive Ray Tracing". In: *J. Syst. Archit.* 59.3 (Mar. 2013), pages 166–175. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2011.06.003. URL: http://dx.doi.org/10.1016/j.sysarc.2011.06.003 (cited on page 34).

[KR09] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Volume 344. John Wiley & Sons, 2009 (cited on page 48).

[KK86] Timothy L. Kay and James T. Kajiya. "Ray Tracing Complex Scenes". In: volume 20. 4. New York, NY, USA: ACM, Aug. 1986, pages 269–278. DOI: 10.1145/15886.15916. URL: http://doi.acm.org/10.1145/15886.15916 (cited on page 34).

[Kég99] Balázs Kégl. "Principal Curves: Learning, Design, and Applications". PhD thesis. Citeseer, 1999 (cited on pages 76, 78).

[Kég+00] Balázs Kégl, Adam Krzyzak, Tamás Linder, and Kenneth Zeger. "Learning and Design of Principal Curves". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 22.3 (2000), pages 281–297 (cited on pages 75, 76).

[KKP05] James Keller, Raghu Krisnapuram, and Nikhil R. Pal. *Fuzzy Models and Algorithms for Pattern Recognition and Image Processing*. Volume 4. Springer, 2005 (cited on page 48).

[KR03] Byungmoon Kim and Jarek Rossignac. "Collision Prediction for Polyhedra Under Screw Motions". In: *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*. SM '03. Seattle, Washington, USA: ACM, 2003, pages 4–10. ISBN: 1-58113-706-0. DOI: 10.1145/781606.781612. URL: http://doi.acm.org/10.1145/781606.781612 (cited on page 30).

[Kim+09] Duksu Kim, Jae-Pil Heo, Jaehyuk Huh, John Kim, and Sung-eui Yoon. "HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs". In: *Computer Graphics Forum*. Volume 28. 7. Wiley Online Library. 2009, pages 1791–1800 (cited on pages 32, 85).

[KHY09] DukSu Kim, Jae-Pil Heo, and Sung-eui Yoon. "PCCD: Parallel Continuous Collision Detection". In: *SIGGRAPH'09: Posters*. ACM. 2009, page 50 (cited on page 31).

[Kim+11] Yong-Joon Kim, Young-Taek Oh, Seung-Hyun Yoon, Myung-Soo Kim, and Gershon Elber. "Coons BVH for Freeform Geometric Models". In: *Proceedings of the 2011 SIGGRAPH Asia Conference*. SA '11. Hong Kong, China: ACM, 2011, 169:1–169:8. ISBN: 978-1-4503-0807-6. DOI: 10.1145/2024156.2024203. URL: http://doi.acm.org/10.1145/2024156.2024203 (cited on page 11).

[KW10] David B. Kirk and W. Hwu Wen-mei. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier, 2010 (cited on page 119).

[KZ03a] Jan Klein and Gabriel Zachmann. "ADB-Trees: Controlling the Error of Time-Critical Collision Detection." In: *VMV*. 2003, pages 37–45 (cited on page 33).

[KZ03b] Jan Klein and Gabriel Zachmann. "Time-critical Collision Detection Using an Average-case Approach". In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. VRST '03. Osaka, Japan: ACM, 2003, pages 22–

31. ISBN: 1-58113-569-6. DOI: 10.1145/1008653.1008660. URL: http://doi.acm.org/10.1145/1008653.1008660 (cited on page 33).

[KZ04a] Jan Klein and Gabriel Zachmann. "Point Cloud Collision Detection". In: *Computer Graphics forum (Proc. EUROGRAPHICS)*. Edited by M.-P. Cani and M. Slater. Volume 23. Grenoble, France, Aug. 2004, pages 567–576 (cited on page 11).

[KZ04b] Jan Klein and Gabriel Zachmann. "Point Cloud Surfaces using Geometric Proximity Graphs". In: *Computers & Graphics* 28.6 (2004), pages 839–850 (cited on page 11).

[KZ04c] Jan Klein and Gabriel Zachmann. "Proximity graphs for defining surfaces over point clouds". In: *Proceedings of the First Eurographics conference on Point-Based Graphics*. Eurographics Association. 2004, pages 131–138 (cited on page 11).

[KZ05] Jan Klein and Gabriel Zachmann. "Interpolation Search for Point Cloud Intersection". In: *Proc. of WSCG 2005*. Edited by Vaclav Skala. University of West Bohemia, Plzeň, Czech Republic, Jan. 2005, pages 163–170. ISBN: 80-903100-7-9 (cited on page 12).

[KT06] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson Education India, 2006 (cited on page 25).

[Klo+98] James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, and Karel Zikan. "Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs". In: *Visualization and Computer Graphics, IEEE Transactions on* 4.1 (1998), pages 21–36 (cited on pages 23, 24).

[Klo98] James Thomas Klosowski. "Efficient Collision Detection for Interactive 3D Graphics and Virtual Environments". PhD thesis. State University of New York, 1998 (cited on page 23).

[Kno+11] A. Knoll, S. Thelen, I. Wald, C.D. Hansen, H. Hagen, and M.E. Papka. "Full-resolution interactive CPU volume rendering with coherent BVH traversal". In: *Pacific Visualization Symposium (PacificVis), 2011 IEEE*. Mar. 2011, pages 3–10. DOI: 10.1109/PACIFICVIS.2011.5742355 (cited on page 36).

[Koc+07] Sinan Kockara, Tansel Halic, K. Iqbal, Coskun Bayrak, and Richard Rowe. "Collision Detection: A Survey". In: *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*. IEEE. 2007, pages 4046–4051 (cited on page 14).

[Koh90] Teuvo Kohonen. "The Self-Organizing Map". In: *Proceedings of the IEEE* 78.9 (1990), pages 1464–1480 (cited on page 75).

[KZ97] Petr Konečný and Karel Zikan. "Lower Bound of Distance in 3D". In: *Proceedings of WSCG*. Volume 3. 1997, pages 640–649 (cited on page 23).

[Kop+12] Daniel Kopta, Thiago Ize, Josef Spjut, Erik Brunvand, Al Davis, and Andrew Kensler. "Fast, Effective BVH Updates for Animated Scenes". In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '12. Costa Mesa, California: ACM, 2012, pages 197–204. ISBN: 978-1-4503-1194-6. DOI: 10.1145/2159616.2159649. URL: http://doi.acm.org/10.1145/2159616.2159649 (cited on page 25).

[Kös+09] Michael Köster, Peter Novák, David Mainzer, and Bernd Fuhrmann. "Two Case Studies for Jazzyk BSM". In: *Proceedings of Agents for Games and Simulations, AGS 2009, AAMAS 2009 co-located workshop*. Volume 5920. LNAI. Springer Verlag, 2009, pages 31–45. URL: http://link.springer.com/book/10.1007/978-3-642-11198-3.

[KP12] Janusz Kowalik and Tadeusz Puźniakowski. *Using OpenCL: Programming Massively Parallel Computers (Advances in Parallel Computing)*. Har/Cdr. Volume 21. IOS Press, Mar. 2012. ISBN: 9781614990291 (cited on pages 40–42, 119).

[Kri+98] Shankar Krishnan, Amol Pattekar, Ming C. Lin, and Dinesh Manocha. "Spherical Shell: A Higher Order Bounding Volume for Fast Proximity Queries". In: WAFR '98 (1998), pages 177–190. URL: http://dl.acm.org/citation.cfm?id=298960.299006 (cited on page 23).

[KPB12] Thomas Kroes, Frits H. Post, and Charl P. Botha. "Exposure Render: An Interactive Photo-Realistic Volume Rendering Framework". In: *PloS one* 7.7 (2012), e38586 (cited on page 36).

[KW03] Jens Krüger and Rüdiger Westermann. "Linear Algebra Operators for GPU Implementation of Numerical Algorithms". In: *ACM SIGGRAPH 2003 Papers*. SIGGRAPH '03. San Diego, California: ACM, 2003, pages 908–916. ISBN: 1-58113-709-5. DOI: 10.1145/1201775.882363. URL: http://doi.acm.org/10.1145/1201775.882363 (cited on page 38).

[KJP02] Paul G. Kry, Doug L. James, and Dinesh K. Pai. "Eigenskin: Real Time Large Deformation Character Skinning in Hardware". In: *Proceedings of the 2002 ACM SIG-*

GRAPH/Eurographics symposium on Computer animation. ACM. 2002, pages 153–159 (cited on pages 18, 78, 105).

[Lar+01] R. Lario, C. Garcia, M. Prieto, and F. Tirado. "Rapid Parallelization of a Multilevel Cloth Simulator Using OpenMP". In: *3rd European Workshop on OpenMP. In Conjuction with IEEE/ACM PACT*. Sept. 2001, page 40. URL: http://artecs.dacya.ucm.es/sites/default/files/ewomp2001_0.pdf (cited on page 31).

[Lar+99] Eric Larsen, Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. *Fast Proximity Queries with Swept Sphere Volumes*. Technical report. 1999. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.8240 (cited on page 92).

[LA01] Thomas Larsson and Tomas Akenine-Möller. "Collision Detection for Continuously Deforming Bodies". In: *Eurographics 2001*. 2001, pages 325–333 (cited on page 23).

[Las87] John Lasseter. "Principles of Traditional Animation applied to 3D Computer Animation". In: *ACM Siggraph Computer Graphics*. Volume 21. 4. ACM. 1987, pages 35–44 (cited on page 17).

[Lau+02] Rynson W. H. Lau, Oliver Chan, Mo Luk, and Frederick W. B. Li. "LARGE a Collision Detection Framework for Deformable Objects". In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. VRST '02. Hong Kong, China: ACM, 2002, pages 113–120. ISBN: 1-58113-530-0. DOI: 10.1145/585740.585760. URL: http://doi.acm.org/10.1145/585740.585760 (cited on page 10).

[LH91] David Laur and Pat Hanrahan. "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering". In: *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '91. New York, NY, USA: ACM, 1991, pages 285–288. ISBN: 0-89791-436-8. DOI: 10.1145/122718.122748. URL: http://doi.acm.org/10.1145/122718.122748 (cited on page 35).

[LMM10] Christian Lauterbach, Qi Mo, and Dinesh Manocha. "gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries". In: *Computer Graphics Forum*. Volume 29. 2. Wiley Online Library. 2010, pages 419–428 (cited on page 32).

[Le 07] Scott Le Grand. "Broad-phase Collision Detection with CUDA". In: *GPU gems 3* (2007), pages 697–721 (cited on page 20).

[Lev+02] Joshua Leven, Jason Corso, Jonathan Cohen, and Subodh Kumar. "Interactive Visualization of Unstructured Grids using Hierarchical 3D Textures". In: *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*. IEEE Press. 2002, pages 37–44 (cited on page 35).

[LCF00] J. P. Lewis, Matt Cordner, and Nickson Fong. "Pose Space Deformation: A Unified Approach to Shape Interpolation and Skeleton-driven Deformation". In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pages 165–172. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344862. URL: http://dx.doi.org/10.1145/344779.344862 (cited on page 78).

[LC96] Chyi-Cheng Lin and Yu-Tai Ching. "An Efficient Volume-Rendering Algorithm with an Analytic Approach". In: *The Visual Computer* 12.10 (1996), pages 515–526 (cited on page 35).

[Lin93] Ming C. Lin. *Efficient Collision Detection for Animation and Robotics*. Technical report. 1993 (cited on pages 20, 29).

[LC91] Ming C. Lin and John F. Canny. "A Fast Algorithm for Incremental Distance Calculation". In: *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*. IEEE. 1991, pages 1008–1014 (cited on page 25).

[LG98] Ming Lin and Stefan Gottschalk. "Collision Detection Between Geometric Models: A Survey". In: *Proc. of IMA conference on mathematics of surfaces*. Volume 1. 1998, pages 602–608 (cited on pages 10, 14).

[Liu+10] Fuchang Liu, Takahiro Harada, Youngeun Lee, and Young J. Kim. "Real-time Collision Culling of a Million Bodies on Graphics Processing Units". In: *ACM Transactions on Graphics (TOG)* 29.6 (2010), page 154 (cited on pages 20, 74).

[Lla+03] Ignacio Llamas, Byungmoon Kim, Joshua Gargus, Jarek Rossignac, and Chris D. Shaw. "Twister: A Space-Warp Operator for the Two-Handed Editing of 3D Shapes". In: *ACM Transactions on Graphics (TOG)*. Volume 22. 3. ACM. 2003, pages 663–668 (cited on page 17).

[LCN99] J.-C. Lombardo, M.-P. Cani, and Fabrice Neyret. "Real-time Collision Detection for Virtual Surgery". In: *Computer Animation, 1999. Proceedings*. IEEE. 1999, pages 82–90 (cited on page 7).

[LC87] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: ACM, 1987, pages 163–169. ISBN: 0-89791-227-6. DOI: 10.1145/37401.37422. URL: http://doi.acm.org/10.1145/37401.37422 (cited on page 35).

[LCF05] Rodrigo G. Luque, João L. D. Comba, and Carla M. D. S. Freitas. "Broad-phase Collision Detection Using Semi-adjusting BSP-trees". In: *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. I3D '05. Washington, District of Columbia: ACM, 2005, pages 179–186. ISBN: 1-59593-013-2. DOI: 10.1145/1053427.1053457. URL: http://doi.acm.org/10.1145/1053427.1053457 (cited on page 20).

[Mac+67] James MacQueen et al. "Some Methods for Classification and Analysis of MultiVariate Observations". In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. 281-297. California, USA. 1967, page 14 (cited on page 50).

[MWZ11] David Mainzer, René Weller, and Gabriel Zachmann. "Kollisionserkennung und natürliche Interaktion in vir"-tu"-ellen Umgebungen". In: *Virtuelle Techniken im industriellen Umfeld*. Edited by Werner Schreiber und Peter Zimmermann. Springer, 2011. Chapter 3.2, 3.4, pages 33–38, 114–116. ISBN: 978-3-642-20635-1. URL: http://www.springer.com/engineering/signals/book/978-3-642-20635-1.

[MZ13] David Mainzer and Gabriel Zachmann. "CDFC: Collision Detection Based on Fuzzy Clustering for Deformable Objects on GPUs". In: *WSCG 2013 - POSTER Proceedings*. Volume 21. 3. Poster. Plzeň, Czech Republic, July 2013, pages 5–8.

[MZ14] David Mainzer and Gabriel Zachmann. "Collision Detection Based on Fuzzy Scene Subdivision". In: *Symposium on GPU Computing and Applications (Singapore, 2013)*. Edited by Yiyu Cai and Simon See. Volume 3. Springer, 2014. URL: https://www.springer.com/engineering/signals/book/978-981-287-133-6.

[Mar+96] William R. Mark, Scott C. Randolph, Mark Finch, James M. Van Verth, and Russell M. Taylor II. "Adding Force Feedback to Graphics Systems: Issues and Solutions". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM. 1996, pages 447–452 (cited on page 116).

[MBS93] Thomas M. Martinetz, Stanislav G. Berkovich, and Klaus J. Schulten. "'Neural-Gas' Network for Vector Quantization and its Application to Time-Series Prediction". In: *Neural Networks, IEEE Transactions on* 4.4 (1993), pages 558–569 (cited on pages 53, 54).

[MRB09] Zoltan Csaba Marton, Radu Bogdan Rusu, and Michael Beetz. "On Fast Surface Reconstruction Methods for Large and Noisy Datasets". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. Kobe, Japan, May 2009 (cited on page 115).

[McA+11] Aleka McAdams, Yongning Zhu, Andrew Selle, Mark Empey, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. "Efficient Elasticity for Character Skinning with Contact and Collisions". In: *ACM SIGGRAPH 2011 Papers*. SIGGRAPH '11. Vancouver, British Columbia, Canada: ACM, 2011, 37:1–37:12. ISBN: 978-1-4503-0943-1. DOI: 10.1145/1964921.1964932. URL: http://doi.acm.org/10.1145/1964921.1964932 (cited on page 18).

[MPT99] William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. "Six Degree-of-freedom Haptic Rendering Using Voxel Sampling". In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pages 401–408. ISBN: 0-201-48560-5 (cited on pages 15, 98).

[Met92] Dimitri N. Metaxas. "Physics-Based Modeling of Nonrigid Objects for Vision and Graphics". PhD thesis. University of Toronto, 1992 (cited on page 78).

[Met96] Dimitris N. Metaxas. *Physics-Based Deformable Models: Applications to Computer Vision, Graphics, and Medical Imaging*. 1st. Norwell, MA, USA: Kluwer Academic Publishers, 1996. ISBN: 0792398408 (cited on page 78).

[Mil+02] Tim Milliron, Robert J. Jensen, Ronen Barzel, and Adam Finkelstein. "A Framework for Geometric Warps and Deformations". In: *ACM Transactions on Graphics (TOG)* 21.1 (2002), pages 20–51 (cited on page 17).

[Mir97] Brian Mirtich. "Efficient Algorithms for Two-Phase Collision Detection". In: *Practical motion planning in robotics: current approaches and future directions* (1997), pages 203–223 (cited on page 19).

[Mir98] Brian Mirtich. "V-Clip: Fast and Robust Polyhedral Collision Detection". In: *ACM Transactions on Graphics (TOG)* 17.3 (1998), pages 177–208 (cited on page 25).

[Miro0] Brian Mirtich. "Timewarp Rigid Body Simulation". In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pages 193–200. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344866. URL: http://dx.doi.org/10.1145/344779.344866 (cited on page 30).

[Mir96] Brian Vincent Mirtich. "Impulse-based Dynamic Simulation of Rigid Body Systems". PhD thesis. University of California, 1996 (cited on page 29).

[MC95] Brian Mirtich and John Canny. "Impulse-based Simulation of Rigid bodies". In: *Proceedings of the 1995 symposium on Interactive 3D graphics*. ACM. 1995, 181–ff (cited on page 29).

[Moh12] Daniel Mohr. "Model-Based High-Dimensional Pose Estimation with Application to Hand Tracking". PhD thesis. Bremen, 2012. URL: http://elib.suub.uni-bremen.de/cgi-bin/diss/user/zsearch?search=sqn&FORMAT=XML&XML_STYLE=/diss/long-DE.xml&userid=nobody&sqn=00102865 (cited on page 116).

[MZ10] Daniel Mohr and Gabriel Zachmann. "Silhouette Area Based Similarity Measure for Template Matching in Constant Time". In: *6th International Conference of Articulated Motion and Deformable Objects (AMDO)*. Port d'Andratx, Mallorca, Spain: Springer Verlag, July 2010, pages 43–54. URL: http://cgvr.cs.uni-bremen.de/research/handtracking/index.shtml (cited on page 33).

[Möl97] Tomas Möller. "A Fast Triangle-Triangle Intersection Test". In: *Journal of graphics tools* 2.2 (1997), pages 25–30 (cited on page 81).

[MEP92] Steven Molnar, John Eyles, and John Poulton. "PixelFlow: High-Speed Rendering Using Image Composition". In: *ACM SIGGRAPH Computer Graphics*. Volume 26. 2. ACM. 1992, pages 231–240 (cited on page 8).

[Moo12] Thomas Mooney. *Unreal Development Kit Game Design Cookbook*. Packt Publishing, Feb. 2012. ISBN: 9781849691802 (cited on page 9).

[MA03] Kenneth Moreland and Edward Angel. "The FFT on a GPU". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '03. San Diego, California: Eurographics Association, 2003, pages 112–119. ISBN: 1-58113-739-7. URL: http://dl.acm.org/citation.cfm?id=844174.844191 (cited on page 38).

[MP78] David E. Muller and Franco P. Preparata. "Finding the Intersection of Two Convex Polyhedra". In: *Theoretical Computer Science* 7.2 (1978), pages 217–236 (cited on pages 5, 8).

[Mül+02] Matthias Müller, Julie Dorsey, Leonard McMillan, Robert Jagnow, and Barbara Cutler. "Stable Real-Time Deformations". In: *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*. ACM. 2002, pages 49–54 (cited on page 18).

[Nay93] Bruce Naylor. "Constructing Good Partitioning Trees". In: *Graphics Interface*. CANADIAN INFORMATION PROCESSING SOCIETY. 1993, pages 181–181 (cited on page 45).

[NAT90] Bruce Naylor, John Amanatides, and William Thibault. "Merging BSP Trees Yields Polyhedral Set Operations". In: *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '90. Dallas, TX, USA: ACM, 1990, pages 115–124. ISBN: 0-89791-344-2. DOI: 10.1145/97879.97892. URL: http://doi.acm.org/10.1145/97879.97892 (cited on page 45).

[Nea+06] Andrew Nealen, Matthias Müller, Richard Keiser, Eddy Boxerman, and Mark Carlson. "Physically Based Deformable Models in Computer Graphics". In: *Computer Graphics Forum*. Volume 25. 4. Wiley Online Library. 2006, pages 809–836 (cited on pages 15, 18).

[Nea+07] Andrew Nealen, Olga Sorkine, Marc Alexa, and Daniel Cohen-Or. "A Sketch-Based Interface for Detail-Preserving Mesh Editing". In: *ACM SIGGRAPH 2007 courses*. ACM. 2007, page 42 (cited on page 17).

[Neu+02] André Neubauer, Lukas Mroz, Helwig Hauser, and Rainer Wegenkittl. "Cell-Based First-Hit Ray Casting". In: *Proceedings of the symposium on Data Visualisation 2002*. 2002, pages 77–86 (cited on page 35).

[Neu45] John von Neumann. "First Draft of a Report on the EDVAC". In: (1945) (cited on page 3).

[Ngu07] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007. ISBN: 9780321545428 (cited on page 3).

[Nvi14] CUDA Nvidia. *CUDA C Programming Guide Version 6.5*. Aug. 2014 (cited on pages 4, 39).

[OH99] James F. O'Brien and Jessica K. Hodgins. "Graphical Modeling and Animation of Brittle Fracture". In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999,

pages 137–146. ISBN: 0-201-48560-5. DOI: 10.1145/311535.311550. URL: http://dx.doi.org/10.1145/311535.311550 (cited on page 18).

[ORo85] Joseph O'Rourke. "Finding Minimal Enclosing Boxes". English. In: *International Journal of Computer & Information Sciences* 14.3 (1985), pages 183–199. ISSN: 0091-7036. DOI: 10.1007/BF00991005. URL: http://dx.doi.org/10.1007/BF00991005 (cited on page 10).

[OD99] Carol O'Sullivan and John Dingliana. *Real-Time Collision Detection and Response Using Sphere-Trees*. 1999 (cited on page 2).

[Ove88] Mark H. Overmars. *Geometric Data Structures for Computer Graphics: An Overview*. Springer, 1988 (cited on pages 14, 27, 28).

[Owe+08] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. "GPU Computing". In: *Proceedings of the IEEE* 96.5 (2008), pages 879–899 (cited on page 3).

[PKS10] Simon Pabst, Artur Koch, and Wolfgang Straßer. "Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces". In: *Computer Graphics Forum*. Volume 29. 5. Wiley Online Library. 2010, pages 1605–1612 (cited on pages 19, 20, 32, 85, 88).

[PG03] Francis Page and Frangois Guibault. "Collision Detection Algorithm for NURBS Surfaces in Interactive Applications". In: *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on*. Volume 2. IEEE. 2003, pages 1417–1420 (cited on page 10).

[PG95] Ian J. Palmer and Richard L. Grimsdale. "Collision Detection for Animation using Sphere-Trees". In: *Computer Graphics Forum*. Volume 14. 2. Wiley Online Library. 1995, pages 105–116 (cited on page 23).

[PCM11] Jia Pan, Sachin Chitta, and Dinesh Manocha. "Probabilistic Collision Detection between Noisy Point Clouds using Robust Classification". In: *International symposium on robotics research*. 2011 (cited on page 12).

[PM12] Jia Pan and Dinesh Manocha. "GPU-based Parallel Collision Detection for Fast Motion Planning". In: *Int. J. Rob. Res.* 31.2 (Feb. 2012), pages 187–200. ISSN: 0278-3649. DOI: 10.1177/0278364911429335. URL: http://dx.doi.org/10.1177/0278364911429335 (cited on page 32).

[Pan+13] Jia Pan, Ioan A. Sucan, Sachin Chitta, and Dinesh Manocha. "Real-time Collision Detection and Distance Computation on Point Cloud Sensor Data". In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, pages 3593–3599 (cited on page 12).

[PRD09] Qi Pan, Gerhard Reitmayr, and Tom Drummond. "ProFORMA: Probabilistic Feature-based On-line Rapid Model Acquisition". In: *Proceedings of the British Machine Vision Conference*. doi:10.5244/C.23.112. BMVA Press, 2009, pages 112.1–112.11. ISBN: 1-901725-39-1 (cited on pages 11, 115).

[Par+10] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. "OptiX: A General Purpose Ray Tracing Engine". In: *ACM SIGGRAPH 2010 Papers*. SIGGRAPH '10. Los Angeles, California: ACM, 2010, 66:1–66:13. ISBN: 978-1-4503-0210-4. DOI: 10.1145/1833349.1778803. URL: http://doi.acm.org/10.1145/1833349.1778803 (cited on pages 34, 38).

[Par+98] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. "Interactive Ray Tracing for Isosurface Rendering". In: *Proceedings of the Conference on Visualization '98*. VIS '98. Research Triangle Park, North Carolina, USA: IEEE Computer Society Press, 1998, pages 233–238. ISBN: 1-58113-106-2. URL: http://dl.acm.org/citation.cfm?id=288216.288266 (cited on page 35).

[Ped05] Witold Pedrycz. *Knowledge-based Clustering: From Data to Information Granules*. John Wiley & Sons, 2005 (cited on pages 49, 50, 75).

[PM99] Dan Pelleg and Andrew Moore. "Accelerating Exact K-means Algorithms with Geometric Reasoning". In: *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '99. San Diego, California, USA: ACM, 1999, pages 277–281. ISBN: 1-58113-143-7. DOI: 10.1145/312129.312248. URL: http://doi.acm.org/10.1145/312129.312248 (cited on page 51).

[Pfi+00] Hanspeter Pfister, Matthias Zwicker, Jeroen Van Baar, and Markus Gross. "Surfels: Surface Elements as Rendering Primitives". In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co. 2000, pages 335–342 (cited on page 11).

[PR69] E. Polak and G. Ribiere. "Note sur la convergence de méthodes de directions conjuguées". In: *ESAIM: Mathematical Modelling and Numerical Analysis - Modélisation Mathématique et Analyse Numérique* 3 (1969), pages 35–43. URL: http://eudml.org/doc/193115 (cited on page 31).

[PML97] Madhav K. Ponamgi, Dinesh Manocha, and Ming C. Lin. "Incremental Algorithms for Collision Detection Between Polygonal Models". In: *IEEE Transactions on Visualization and Computer Graphics* 3.1 (Jan. 1997), pages 51–64. ISSN: 1077-2626. DOI: 10.1109/2945.582346. URL: http://dx.doi.org/10.1109/2945.582346 (cited on page 19).

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry. Texts and Monographs in Computer Science*. 1985 (cited on page 27).

[Pre+07] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3rd edition. Cambridge University Press, Sept. 2007. ISBN: 9780521880688 (cited on pages 74, 79).

[Pro97] Xavier Provot. "Collision and Self-Collision Handling in Cloth Model Dedicated to Design Garments". In: *Computer Animation and Simulation '97*. Edited by Daniel Thalmann and Michiel van de Panne. Eurographics. Springer Vienna, 1997, pages 177–189. ISBN: 978-3-211-83048-2. DOI: 10.1007/978-3-7091-6874-5_13. URL: http://dx.doi.org/10.1007/978-3-7091-6874-5_13 (cited on page 15).

[Pur+02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. "Ray Tracing on Programmable Graphics Hardware". In: *ACM Transactions on Graphics (TOG)*. Volume 21. 3. ACM. 2002, pages 703–712 (cited on page 38).

[ROW14] Mohamed Radwan, Stefan Ohrhallinger, and Michael Wimmer. "Efficient Collision Detection While Rendering Dynamic Point Clouds". In: *Proceedings of the 2014 Graphics Interface Conference*. Montreal, Quebec, Canada, May 2014, pages 25–33. URL: http://www.cg.tuwien.ac.at/research/publications/2014/Radwan-2014-CDR/ (cited on page 12).

[RKC00] Stephane Redon, Abderrahmane Kheddar, and Sabine Coquillart. "An Algebraic Solution to the Problem of Collision Detection for Rigid Polyhedral Objects". In: *Proc. of IEEE Conference on Robotics and Automation*. Citeseer. 2000 (cited on page 30).

[RKC02] Stephane Redon, Abderrahmane Kheddar, and Sabine Coquillart. "Fast Continuous Collision Detection between Rigid Bodies." In: *Computer Graphics Forum (Proc. of EUROGRAPHICS 2002)* 21.3 (2002) (cited on pages 10, 30).

[RSH00] Erik Reinhard, Brian Smits, and Charles Hansen. *Dynamic Acceleration Structures for Interactive Ray Tracing*. Springer, 2000 (cited on page 34).

[Ren+10] Qian Ren, Dongmei Wu, Shuguo Wang, Yili Fu, and Hegao Cai. "Collision Detection Algorithm in Virtual Environment of Robot Workcell". English. In: *Artificial Intelligence and Computational Intelligence*. Edited by FuLee Wang, Hepu Deng, Yang Gao, and Jingsheng Lei. Volume 6319. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pages 292–300. ISBN: 978-3-642-16529-0. DOI: 10.1007/978-3-642-16530-6_35. URL: http://dx.doi.org/10.1007/978-3-642-16530-6_35 (cited on page 7).

[RLN06] Taehyun Rhee, John P. Lewis, and Ulrich Neumann. "Real-Time Weighted Pose-Space Deformation on the GPU". In: *Computer Graphics Forum*. Volume 25. 3. Wiley Online Library. 2006, pages 439–448 (cited on pages 78, 105).

[RB92] Elon Rimon and Stephen P. Boyd. "Efficient Distance Computation Using Best Ellipsoid Fit". In: *In Proceedings of the 1992 IEEE Symposium on Intelligence Control*. Citeseer. 1992 (cited on page 23).

[Rit90] Jack Ritter. "An Efficient Bounding Sphere". In: *Graphics gems*. Academic Press Professional, Inc. 1990, pages 301–303 (cited on page 23).

[RHL02] Szymon Rusinkiewicz, Olaf Hall-Holt, and Marc Levoy. "Real-time 3D Model Acquisition". In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '02. San Antonio, Texas: ACM, 2002, pages 438–446. ISBN: 1-58113-521-1. DOI: 10.1145/566570.566600. URL: http://doi.acm.org/10.1145/566570.566600 (cited on pages 11, 115).

[RL00] Szymon Rusinkiewicz and Marc Levoy. "QSplat: A Multiresolution Point Rendering System for Large Meshes". In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pages 343–352. ISBN: 1-58113-208-5. DOI: 10.1145/344779.344940. URL: http://dx.doi.org/10.1145/344779.344940 (cited on page 11).

[Sag+08] M. Sagardia, T. Hulin, C. Preusche, and G. Hirzinger. "Improvements of the Voxmap-PointShell Algorithm-Fast Generation of Haptic Data-Structures". In: *53rd IWK-Internationales Wissenschaftliches Kolloquium, Ilmenau, Germany*. 2008 (cited on page 98).

[Sam84] Hanan Samet. "The Quadtree and Related Hierarchical Data Structures". In: *ACM Computing Surveys (CSUR)* 16.2 (1984), pages 187–260 (cited on page 24).

[SHG09] Nadathur Satish, Mark Harris, and Michael Garland. "Designing Efficient Sorting Algorithms for Manycore GPUs". In: *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*. May 2009 (cited on pages 58, 59).

[Sch+00] Elmar Schömer, Jürgen Sellen, Marek Teichmann, and Chee Yap. "Smallest Enclosing Cylinders". In: *Algorithmica* 27.2 (2000), pages 170–186 (cited on page 23).

[Sed+04] Thomas W. Sederberg, David L. Cardon, G. Thomas Finnigan, Nicholas S. North, Jianmin Zheng, and Tom Lyche. "T-spline Simplification and Local Refinement". In: *ACM SIGGRAPH 2004 Papers*. SIGGRAPH '04. Los Angeles, California: ACM, 2004, pages 276–283. DOI: 10.1145/1186562.1015715. URL: http://doi.acm.org/10.1145/1186562.1015715 (cited on page 17).

[Sed+03] Thomas W. Sederberg, Jianmin Zheng, Almaz Bakenov, and Ahmad Nasri. "T-splines and T-NURCCs". In: *ACM transactions on graphics (TOG)*. Volume 22. 3. ACM. 2003, pages 477–484 (cited on page 17).

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms (4th Edition)*. 4th. Addison-Wesley Professional, Mar. 2011. ISBN: 9780321573513 (cited on page 28).

[Sel+09] Andrew Selle, Jonathan Su, Geoffrey Irving, and Ronald Fedkiw. "Robust High-Resolution Cloth Using Parallelism, History-Based Collisions, and Accurate Friction". In: *IEEE Transactions on Visualization and Computer Graphics* 15.2 (Mar. 2009), pages 339–350. ISSN: 1077-2626. DOI: 10.1109/TVCG.2008.79. URL: http://dx.doi.org/10.1109/TVCG.2008.79 (cited on page 31).

[SHG08] Shubhabrata Sengupta, Mark Harris, and Michael Garland. *Efficient Parallel Scan Algorithms for GPUs*. Technical report NVR-2008-003. NVIDIA Corporation, Dec. 2008. URL: http://mgarland.org/papers.html%5C#segscan-tr (cited on pages 58, 59).

[She14] Evan Shellshear. "1D Sweep-and-Prune Self-Collision Detection for Deforming Cables". In: *The Visual Computer* 30.5 (2014), pages 553–564. DOI: 10.1007/s00371-013-0880-7. URL: http://dx.doi.org/10.1007/s00371-013-0880-7 (cited on page 20).

[SSK07] Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. "Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes". In: *Computer Graphics Forum*. Volume 26. 3. Wiley Online Library. 2007, pages 395–404 (cited on page 47).

[SAM09] Peter Shirley, Michael Ashikhmin, and Steve Marschner. *Fundamentals of Computer Graphics*. 3rd edition. A K Peters/CRC Press, July 2009. ISBN: 9781568814698 (cited on page 2).

[SW82] Hans-Werner Six and Derick Wood. "Counting and Reporting Intersections of d-Ranges". In: *Computers, IEEE Transactions on* 100.3 (1982), pages 181–187 (cited on page 27).

[Smi+95] Andrew Smith, Yoshifumi Kitamura, Haruo Takemura, and Fumio Kishino. "A Simple and Efficient Method for Accurate Collision Detection Among Deformable Polyhedral Objects in Arbitrary Motion". In: *Proceedings of the Virtual Reality Annual International Symposium (VRAIS'95)*. VRAIS '95. Washington, DC, USA: IEEE Computer Society, 1995, pages 136–. ISBN: 0-8186-7084-3. URL: http://dl.acm.org/citation.cfm?id=527216.836015 (cited on page 25).

[Sny95] John M. Snyder. "An Interactive Tool for Placing Curved Surfaces Without Interpenetration". In: *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '95. New York, NY, USA: ACM, 1995, pages 209–218. ISBN: 0-89791-701-4. DOI: 10.1145/218380.218444. URL: http://doi.acm.org/10.1145/218380.218444 (cited on page 30).

[Sny+93] John M. Snyder, Adam R. Woodbury, Kurt Fleischer, Bena Currin, and Alan H. Barr. "Interval Methods for Multi-Point Collisions between Time-Dependent Curved Surfaces". In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM. 1993, pages 321–334 (cited on page 30).

[SL98] John Snyder and Jed Lengyel. "Visibility Sorting and Compositing Without Splitting for Image Layer Decompositions". In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pages 219–230. ISBN:

0-89791-999-8. DOI: 10.1145/280814.280878. URL: http://doi.acm.org/10.1145/280814.280878 (cited on page 20).

[Sor+04] O. Sorkine, D. Cohen-Or, Y. Lipman, M. Alexa, C. Rössl, and H.-P. Seidel. "Laplacian Surface Editing". In: *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*. SGP '04. Nice, France: ACM, 2004, pages 175–184. ISBN: 3-905673-13-4. DOI: 10.1145/1057432.1057456. URL: http://doi.acm.org/10.1145/1057432.1057456 (cited on page 17).

[STH13] Jonas Spillmann, Stefan Tuchschmid, and Matthias Harders. "Adaptive Space Warping to Enhance Passive Haptics in an Arthroscopy Surgical Simulator". In: *Visualization and Computer Graphics, IEEE Transactions on* 19.4 (2013), pages 626–633. ISSN: 1077-2626. DOI: 10.1109/TVCG.2013.23 (cited on page 17).

[SLY96] C. J. Su, F. H. Lin, and B. P. Yen. "An Adaptive Bounding Object Based Algorithm For Efficient And Precise Collision Detection Of CSG-Represented Virtual Objects". In: *Proc. of Symposium on virtual reality in manufacturing research and education*. 1996 (cited on page 11).

[SLY99] Chuan-Jun Su, Fuhua Lin, and Lan Ye. "A New Collision Detection Method for CSG-represented Objects in Virtual Manufacturing". In: *Comput. Ind.* 40.1 (Sept. 1999), pages 1–13. ISSN: 0166-3615. DOI: 10.1016/S0166-3615(99)00010-X. URL: http://dx.doi.org/10.1016/S0166-3615(99)00010-X (cited on page 11).

[SL05] Herb Sutter and James Larus. "Software and the Concurrency Revolution". In: *Queue* 3.7 (Sept. 2005), pages 54–62. ISSN: 1542-7730. DOI: 10.1145/1095408.1095421. URL: http://doi.acm.org/10.1145/1095408.1095421 (cited on page 37).

[ST92] Richard Szeliski and David Tonnesen. *Surface Modeling with Oriented Particle Systems*. Volume 26. 2. ACM, 1992 (cited on page 17).

[TK06] Hiroyuki Takizawa and Hiroaki Kobayashi. "Hierarchical Parallel Processing of Large Scale Data Clustering on a PC Cluster with GPU Co-processing". In: *The Journal of Supercomputing* 36.3 (June 2006), pages 219–234. ISSN: 0920-8542. DOI: 10.1007/s11227-006-8294-1. URL: http://dx.doi.org/10.1007/s11227-006-8294-1 (cited on page 51).

[Tan+09] Min Tang, Sean Curtis, S.-E. Yoon, and Dinesh Manocha. "ICCD: Interactive Continuous Collision Detection between Deformable Models using Connectivity-Based Culling". In: *Visualization and Computer Graphics, IEEE Transactions on* 15.4 (2009), pages 544–557 (cited on page 10).

[Tan+11] Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. "Collision-streams: Fast GPU-based Collision Detection for Deformable Models". In: *Symposium on Interactive 3D Graphics and Games*. I3D '11. San Francisco, California: ACM, 2011, pages 63–70. ISBN: 978-1-4503-0565-5. DOI: 10.1145/1944745.1944756. URL: http://doi.acm.org/10.1145/1944745.1944756 (cited on pages 32, 85, 87, 88).

[TMT09] Min Tang, Dinesh Manocha, and Ruofeng Tong. "Multi-Core Collision Detection between Deformable Models". In: *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*. ACM. 2009, pages 355–360 (cited on page 31).

[TMT10] Min Tang, Dinesh Manocha, and Ruofeng Tong. "MCCD: Multi-Core Collision Detection between Deformable Models using Front-Based Decomposition". In: *Graphical Models* 72.2 (2010), pages 7–23 (cited on pages 31, 85).

[TF88] Demetri Terzopoulos and Kurt Fleischer. "Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture". In: *ACM Siggraph Computer Graphics*. Volume 22. 4. ACM. 1988, pages 269–278 (cited on page 17).

[Ter+87] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. "Elastically Deformable Models". In: *ACM Siggraph Computer Graphics*. Volume 21. 4. ACM. 1987, pages 205–214 (cited on pages 15, 17).

[Tes+04] Matthias Teschner, Bruno Heidelberger, Matthias Müller, and Markus Gross. "A Versatile and Robust Model for Geometrically Complex Deformable Solids". In: *Proceedings of the Computer Graphics International*. CGI '04. Washington, DC, USA: IEEE Computer Society, 2004, pages 312–319. ISBN: 0-7695-2171-1. DOI: 10.1109/CGI.2004.6. URL: http://dx.doi.org/10.1109/CGI.2004.6 (cited on page 10).

[Tes+03] Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomerantes, and Markus H. Gross. "Optimized Spatial Hashing for Collision Detection of Deformable Objects." In: *Proc. 8th International Fall Workshop Vision, Modeling, and Visualization (VMV 2003)*. 2003, pages 47–54 (cited on page 10).

[Tes+05] Matthias Teschner, Stefan Kimmerle, Bruno Heidelberger, Gabriel Zachmann, Laks Raghupathi, Arnulph Fuhrmann, M.-P. Cani, François Faure, Nadia Magnenat-Thalmann, Wolfgang Strasser, and Pascal Volino. "Collision Detection for Deformable Objects". In: *Computer Graphics Forum*. Volume 24. 1. Wiley Online Library. 2005, pages 61–81 (cited on page 14).

[TR98] David G. Thaler and Chinya V. Ravishankar. "Distributed Top-Down Hierarchy Construction". In: *In Proc. of the IEEE INFOCOM*. 1998, pages 693–701 (cited on page 24).

[TN87] William C. Thibault and Bruce F. Naylor. "Set Operations on Polyhedra using Binary Space Partitioning Trees". In: *ACM SIGGRAPH computer graphics*. Volume 21. 4. ACM. 1987, pages 153–162 (cited on page 20).

[TB07] Bernhard Thomaszewski and Wolfgang Blochinger. "Physically Based Simulation of Cloth on Distributed Memory Architectures". In: *Parallel Comput.* 33.6 (June 2007), pages 377–390. ISSN: 0167-8191. DOI: 10.1016/j.parco.2007.02.008. URL: http://dx.doi.org/10.1016/j.parco.2007.02.008 (cited on page 31).

[TPB08] Bernhard Thomaszewski, Simon Pabst, and Wolfgang Blochinger. "Special Section: Parallel Graphics and Visualization: Parallel Techniques for Physically Based Simulation on Multi-core Processor Architectures". In: *Computers & Graphics* 32.1 (Feb. 2008), pages 25–40. ISSN: 0097-8493 (cited on page 31).

[Til84] Robert B. Tilove. "A Null-Object Detection Algorithm for Constructive Solid Geometry". In: *Communications of the ACM* 27.7 (1984), pages 684–694 (cited on page 11).

[Tor90] Enric Torres. "Optimization of the Binary Space Partition Algorithm (BSP) for the Visualization of Dynamic Scenes". In: *Eurographics*. Volume 90. 1990, pages 507–518 (cited on page 20).

[TWZ07] Sven Trenkel, René Weller, and Gabriel Zachmann. "A Benchmarking Suite for Static Collision Detection Algorithms". In: *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*. Edited by Václav Skala. Plzeň, Czech Republic: Union Agency, 29 January–1 February 2007. URL: http://cg.in.tu-clausthal.de/research/colldet_benchmark (cited on pages 89, 91).

[Tsi+07] Nicolas Tsingos, Carsten Dachsbacher, Sylvain Lefebvre, and Matteo Dellepiane. "Instant Sound Scattering". In: *Proceedings of the 18th Eurographics Conference on Render-*

ing Techniques. EGSR'07. Grenoble, France: Eurographics Association, 2007, pages 111–120. ISBN: 978-3-905673-52-4. DOI: 10.2312/EGWR/EGSR07/111-120. URL: http://dx.doi.org/10.2312/EGWR/EGSR07/111-120 (cited on page 33).

[VB04] Gino Van Den Bergen and Gino Johannes Apolonia van den Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann, 2004 (cited on pages 2, 14, 17).

[Van94] George Vaněkčkek. "Back-face Culling applied to Collision Detection of Polyhedra". In: *The Journal of Visualization and Computer Animation* 5.1 (1994), pages 55–63 (cited on page 33).

[Våp+13] Cecilie Våpenstad, Erlend Fagertun Hofstad, Thomas Langø, Ronald Mårvik, and Magdalena Karolina Chmarra. "Perceiving Haptic feedback in Virtual Reality Simulators". In: *Surgical endoscopy* 27.7 (2013), pages 2391–2397 (cited on page 115).

[VT94] Pascal Volino and Nadia Magnenat Thalmann. "Efficient Self-Collision Detection on Smoothly Discretized Surface Animations using Geometrical Shape Regularity". In: *Computer Graphics Forum*. Volume 13. 3. Wiley Online Library. 1994, pages 155–166 (cited on page 4).

[VBZ90] Brian Von Herzen, Alan H. Barr, and Harold R. Zatz. "Geometric Collisions for Time-Dependent Parametric Surfaces". In: *ACM SIGGRAPH Computer Graphics* 24.4 (1990), pages 39–48 (cited on page 30).

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. "Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies". In: *ACM Transactions on Graphics (TOG)* 26.1 (2007), page 6 (cited on page 34).

[Wal+01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. "Interactive Rendering with Coherent Ray Tracing". In: *Computer graphics forum*. Volume 20. 3. Wiley Online Library. 2001, pages 153–165 (cited on page 34).

[WS04] Michael Wand and Wolfgang Straßer. "Multi-resolution Sound Rendering". In: *ACM SIGGRAPH 2004 Sketches*. SIGGRAPH '04. Los Angeles, California: ACM, 2004, pages 47–. ISBN: 1-58113-896-2. DOI: 10.1145/1186223.1186282. URL: http://doi.acm.org/10.1145/1186223.1186282 (cited on page 33).

[Wan+04] Wenping Wang, Yi-King Choi, Bin Chan, Myung-Soo Kim, and Jiaye Wang. "Efficient Collision Detection for Moving Ellipsoids Using Separating Planes". In: *Computing* 72.1-2 (Apr. 2004), pages 235–246. ISSN: 0010-485X. DOI: 10.1007/s00607-003-0060-0. URL: http://dx.doi.org/10.1007/s00607-003-0060-0 (cited on page 23).

[WWK01] Wenping Wang, Jiaye Wang, and Myung-Soo Kim. "An Algebraic Condition for the Separation of Two Ellipsoids". In: *Comput. Aided Geom. Des.* 18.6 (July 2001), pages 531–539. ISSN: 0167-8396. DOI: 10.1016/S0167-8396(01)00049-8. URL: http://dx.doi.org/10.1016/S0167-8396(01)00049-8 (cited on page 23).

[WP02] Xiaohuan Corina Wang and Cary Phillips. "Multi-weight Enveloping: Least-squares Approximation Techniques for Skin Animation". In: *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA '02. San Antonio, Texas: ACM, 2002, pages 129–138. ISBN: 1-58113-573-4. DOI: 10.1145/545261.545283. URL: http://doi.acm.org/10.1145/545261.545283 (cited on page 18).

[Web03] Andrew R. Webb. *Statistical Pattern Recognition*. John Wiley & Sons, 2003 (cited on pages 49, 50).

[WZ97] R. Weber and P. Zezula. "The Theory and Practice of Searches in High Dimensional Dataspaces". In: *Proceedings of the Fourth DELOS Workshop on ImageIndexing and Retrieval*. 1997 (cited on page 51).

[WHG84] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. "Improved Computational Methods for Ray Tracing". In: *ACM Transactions on Graphics (TOG)* 3.1 (1984), pages 52–69 (cited on page 24).

[Wei86] Jerry Weil. "The Synthesis of Cloth Objects". In: *ACM Siggraph Computer Graphics* 20.4 (1986), pages 49–54 (cited on page 15).

[Wel12] René Weller. "New Geometric Data Structures for Collision Detection". PhD Dissertation. Faculty of Computer Science: University of Bremen, Germany, Aug. 2012 (cited on page 9).

[WFZ13] René Weller, Udo Frese, and Gabriel Zachmann. "Parallel Collision Detection in Constant Time". In: *Virtual Reality Interactions and Physical Simulations (VRIPhys)*. Lille, France: Eurographics Association, Nov. 2013 (cited on page 11).

[WKZ06] René Weller, Jan Klein, and Gabriel Zachmann. "A Model for the Expected Running Time of Collision Detection using AABB Trees". In: *Eurographics Symposium on Virtual Environments (EGVE)*. Edited by Roger Hubbold and Ming Lin. Lisbon, Portugal, Aug. 2006 (cited on page 11).

[Wel+10] René Weller, David Mainzer, Mikel Sagardia, Thomas Hulin, Gabriel Zachmann, and Carsten Preusche. "A benchmarking suite for 6-DOF real time collision response algorithms". In: *Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology (VRST)*. Hong Kong: ACM, Nov. 2010, pages 63–70. ISBN: 978-1-4503-0441-2. DOI: http://doi.acm.org/10.1145/1889863.1889874. URL: http://cg.in.tu-clausthal.de/publications.shtml%5C#vrst2010 (cited on page 89).

[Wel+14] René Weller, David Mainzer, Abhishek Srinivas, Matthias Teschner, and Gabriel Zachmann. "Massively Parallel Batch Neural Gas for Bounding Volume Hierarchy Construction". In: *Virtual Reality Interactions and Physical Simulations (VRIPhys)*. Bremen, Germany: Eurographics Association, Sept. 2014 (cited on pages 38, 58).

[WZ12] Rene Weller and Gabriel Zachmann. "User Performance in Complex Bi-manual Haptic Manipulation with 3 DOFs vs. 6 DOFs". In: *Haptics Symposium*. Vancouver, Canada, Mar. 2012. URL: http://cg.in.tu-clausthal.de/research/haptesha/index.shtml (cited on page 116).

[WZ06] René Weller and Gabriel Zachmann. "Kinetic Separation Lists for Continuous Collision Detection of Deformable Objects." In: *VRIPHYS*. 2006, pages 33–42 (cited on page 30).

[WZ09] René Weller and Gabriel Zachmann. "Inner Sphere Trees for Proximity and Penetration Queries". In: *2009 Robotics: Science and Systems Conference (RSS)*. Seattle, WA, USA, June 2009. URL: http://cgvr.cs.uni-bremen.de/research/ist/index.shtml (cited on pages 10, 54, 92, 99).

[WZ10] René Weller and Gabriel Zachmann. "ProtoSphere: A GPU-Assisted Prototype-Guided Sphere Packing Algorithm for Arbitrary Objects". In: *ACM SIGGRAPH ASIA 2010 Sketches*. Seoul, Republic of Korea: ACM, Dec. 2010, 8:1–8:2. ISBN: 978-1-4503-0523-5. DOI: http://doi.acm.org/10.1145/1899950.1899958. URL: http://cg.in.tu-clausthal.de/research/protosphere (cited on page 38).

[Whi80] Turner Whitted. "An Improved Illumination Model for Shaded Display". In: *Commun. ACM* 23.6 (June 1980), pages 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: http://doi.acm.org/10.1145/358876.358882 (cited on page 33).

[Wil13] Nicholas Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013 (cited on page 3).

[WB05] Wingo Sai-Keung Wong and George Baciu. "GPU-based Intrinsic Collision Detection for Deformable Surfaces: Collision Detection and Deformable Objects". In: *Comput. Animat. Virtual Worlds* 16.3-4 (July 2005), pages 153–161. ISSN: 1546-4261. DOI: 10.1002/cav.v16:3/4. URL: http://dx.doi.org/10.1002/cav.v16:3/4 (cited on page 32).

[WDM07] Muiris Woulfe, John Dingliana, and Michael Manzke. "Hardware Accelerated Broad Phase Collision Detection for Realtime Simulations". In: *Workshop in Virtual Reality Interactions and Physical Simulation*. The Eurographics Association. 2007, pages 79–88 (cited on page 19).

[Wu12] Junjie Wu. *Advances in K-means Clustering: A Data Mining Thinking*. Springer, 2012 (cited on page 51).

[WZH09] Ren Wu, Bin Zhang, and Meichun Hsu. "Clustering Billions of Data Points Using GPUs". In: *Proceedings of the Combined Workshops on UnConventional High Performance Computing Workshop Plus Memory Access Workshop*. UCHPC-MAW '09. Ischia, Italy: ACM, 2009, pages 1–6. ISBN: 978-1-60558-557-4. DOI: 10.1145/1531666.1531668. URL: http://doi.acm.org/10.1145/1531666.1531668 (cited on page 52).

[Wu92] Xiaolin Wu. "Graphics Gems III". In: edited by David Kirk. San Diego, CA, USA: Academic Press Professional, Inc., 1992. Chapter A Linear-time Simple Bounding Volume Algorithm, pages 301–306. ISBN: 0-12-409671-9. URL: http://dl.acm.org/citation.cfm?id=130745.130796 (cited on page 74).

[YG07] Türker Yılmaz and Uğur Güdükbay. "Conservative Occlusion Culling for Urban Visualization using a Slice-wise Data Structure". In: *Graphical Models* 69.3 (2007), pages 191–210 (cited on page 33).

[YW93] J.-H. Youn and K. Wohn. "Realtime collision detection for virtual reality applications". In: *Virtual Reality Annual International Symposium, 1993., 1993 IEEE*. Sept. 1993, pages 415–421. DOI: 10.1109/VRAIS.1993.380750 (cited on page 7).

[Yu+04] Yizhou Yu, Kun Zhou, Dong Xu, Xiaohan Shi, Hujun Bao, Baining Guo, and Heung-Yeung Shum. "Mesh Editing with Poisson-based Gradient Field Manipulation". In: *ACM SIGGRAPH 2004 Papers*. SIGGRAPH '04. Los Angeles, California: ACM, 2004, pages 644–651. DOI: 10.1145/1186562.1015774. URL: http://doi.acm.org/10.1145/1186562.1015774 (cited on page 17).

[Zac95] Gabriel Zachmann. "The BoxTree: Exact and Fast Collision Detection of Arbitrary Polyhedra". In: *Informal Proc. First Workshop on Simulation and Interaction in Virtual Environments, SIVE* 95 (1995), pages 104–112 (cited on page 24).

[Zac98a] Gabriel Zachmann. "Rapid Collision Detection by Dynamically Aligned DOP-Trees". In: *Proceedings of the Virtual Reality Annual International Symposium*. VRAIS '98. Washington, DC, USA: IEEE Computer Society, 1998, pages 90–. ISBN: 0-8186-8362-7. URL: http://dl.acm.org/citation.cfm?id=522258.836122 (cited on page 23).

[Zac98b] Gabriel Zachmann. "Rapid Collision Detection by Dynamically Aligned DOP-Trees". In: *Proc. of IEEE Virtual Reality Annual International Symposium; VRAIS '98*. Atlanta, Georgia, Mar. 1998, pages 90–97 (cited on page 24).

[Zac00] Gabriel Zachmann. "Virtual Reality in Assembly Simulation – Collision Detection, Simulation Algorithms, and Interaction Techniques". PhD thesis. Darmstadt University of Technology, Germany, May 2000. ISBN: 3-8167-5628-X (cited on pages 8, 28).

[Zac01] Gabriel Zachmann. "Optimizing the Collision Detection Pipeline". In: *Proc. of the First International Game Technology Conference (GTEC)*. Jan. 2001 (cited on pages 9, 21, 30).

[ZL03] Gabriel Zachmann and Elmar Langetepe. *Geometric Data Structures for Computer Graphics*. Eurographics Assoc., 2003 (cited on page 14).

[ZH97] Hansong Zhang and Kenneth E. Hoff III. "Fast Backface Culling Using Normal Masks". In: *Proceedings of the 1997 symposium on Interactive 3D graphics*. ACM. 1997, 103–ff (cited on page 33).

[Zha+97] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff III. "Visibility Culling Using Hierarchical Occlusion Maps". In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pages 77–88. ISBN: 0-89791-896-7. DOI: 10.1145/258734.258781.

URL: http://dx.doi.org/10.1145/258734.258781 (cited on page 33).

[Zho+11] Kun Zhou, Minmin Gong, Xin Huang, and Baining Guo. "Data-Parallel Octrees for Surface Reconstruction". In: *IEEE Transactions on Visualization and Computer Graphics* 17.5 (May 2011), pages 669–681. ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.75. URL: http://dx.doi.org/10.1109/TVCG.2010.75 (cited on page 19).

[Zho+08] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. "Real-time KD-tree Construction on Graphics Hardware". In: *ACM SIGGRAPH Asia 2008 Papers*. SIGGRAPH Asia '08. Singapore: ACM, 2008, 126:1–126:11. ISBN: 978-1-4503-1831-0. DOI: 10.1145/1457515.1409079. URL: http://doi.acm.org/10.1145/1457515.1409079 (cited on pages 19, 47).

[Zwi+02] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. "EWA Splatting". In: *Visualization and Computer Graphics, IEEE Transactions on* 8.3 (2002), pages 223–238 (cited on page 11).

GLOSSARY

AABB	Axis Aligned Bounding Box. 10–12, 19, 22, 23, 25–28, 30, 74, 79, 86, 89, 92, 94
AMD	Advanced Micro Devices. 83
AoS	Array of Structures. 79
AVX	Advanced Vector Extensions. 63
BNG	Batch Neural Gas. 38, 45, 53–61, 64, 72, 106
BSP	Binary Space Partitioning. 19–21, 45, 46
BV	Bounding Volume. 11, 15, 18–28, 30–34, 55, 57, 64, 73–75, 79–81, 83, 87, 92, 99, 106, 107, 109, 111
BVH	Bounding Volume Hierarch. ix, 7, 10–12, 22–25, 31, 32, 34, 36, 54–58, 62, 63, 90, 113
CAD	Computer-Aided Design. 45
CPU	Central Processing Unit. 2–5, 12, 20, 31, 32, 35–37, 42, 55, 60, 62, 63, 83–85, 88, 97, 100, 101, 111, 114
CSG	Constructive Solid Geometry. 2, 11, 14, 20
CT	Computed Tomography. 35
CUDA	Compute Unified Device Architecture. 20, 37–39, 42, 55, 58, 60, 76, 78–80, 84, 85, 109, 110, 119
DAC	Divide-and-Conquer. 11, 52
DAG	Directed Acyclic Graph. 16
DoF	Degree of Freedom. 110, 116
FDH	Fixed-Direction Hull. 23
FEM	Finite Element Method. 18
FFT	Fast Fourier Transform. 38
FPGA	Field-Programmable Gate Array. 19
GJK	Gilbert-Johnson-Keerthi. 26
GPGPU	General Purpose Computation on GPUs. 37, 38
GPU	Graphics Processing Unit. ix, 2–6, 11, 12, 17, 20, 31, 32, 34–39, 42, 52, 55, 58, 60, 62, 64, 65, 78, 82–85, 88, 106, 107, 111, 112, 114, 119, 161

GUI	Graphical User Interface. 109
IA	Interval Arithmetic. 10, 30
IST	Inner Sphere Tree. 10, 11, 54–57, 92, 97–102, 104, 113
k-DOP	k-Discrete Oriented Polytopes. 23, 25
MIMD	Multiple-Instruction, Multiple-Data. 2, 3, 31
MISD	Multiple-Instruction, Single-Data. 2, 3
MMX	Multi Media Extension. 2
MPR	Multilevel Polak-Ribiere. 31
MRI	Magnetic Resonance Imaging. 35
NG	Neural Gas. 53
NP	Non-Deterministic Polynomial Time. 45
NURBS	Non-Uniform Rational B-Spline. 2, 10, 11, 54
OBB	Oriented Bounding Box. 10, 19, 23, 26, 30, 32, 74
OpenCL	Open Computing Language. 37, 38, 42, 109, 119
OpenMP	Open Multi-Processing. 31
PCA	Principal Component Analysis. 5, 20, 68, 71, 73–76, 78, 79, 86, 106, 113, 114
PCS	Potentially Collision Set. 9, 21
PDE	Partial Differential Equation. 18
PQP	Proximity Query Package. 92
QuOSPO	Quantized Orientation Slabs with Primary Orientations. 23
SAH	Surface Area Heuristic. 34
SaP	Sweep-and-Prune. 5, 7, 20, 21, 26, 27, 65, 73–75, 77–79, 107, 113, 114
SDK	Software Development Kit. 110
SIMD	Single-Instruction, Multiple-Data. 2, 3, 34, 61, 163
SISD	Single-Instruction, Single-Data. 3
SLI	Scalable Link Interface. 83
SMX	Next Generation Streaming Multiprocessor. 38, 39

SoA	Structure of Arrays. 79
SOFA	Simulation Open Framework Architecture. 9
SOM	Self-Organizing Map. 75
SP	Streaming Processor. 39
SPH	Smoothed Particle Hydrodynamics. 18, 36
SPMD	Single-Program, Multiple-Data. 31
SSE	Streaming SIMD Extensions. 2, 63
SSL	Sphere-Swept Line. 23
SSP	Sphere-Swept Point. 23
SSR	Sphere-Swept Rectangle. 23, 32
ToI	Time of Impact. 13
VPS	Voxmap-Pointshell. 97, 98, 100–102, 104